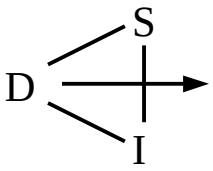


Fondamenti di Informatica
Parte 3
"Il C+ di P. Nesi"

CCL - Amb-Ter

Facoltà di Ingegneria
Università degli Studi di Firenze

Ver.: 1.0
Data: 20/3/2000

Questo documento fa parte di una serie di dispense prodotte dallo sforzo volontario di molti ma che sono state riviste e corrette dal docente solo in parte. Nonostante la loro incompiutezza ho deciso di distribuirle per dare comunque una traccia sugli argomenti del corso in mancanza di un libro di riferimento. Non rappresentano pertanto ne' il libro ne' il programma di riferimento del corso, a questo fine fanno fede le lezioni stesse che in molte parti differiranno da queste dispense.
Si prega pertanto di segnalare ogni mancanza e correzione inviando una mail al Prof. P. Nesi al seguente indirizzo di posta elettronica con soggetto "dispense": nesi@dsi.unifi.it , con la speranza di arrivare a produrre una versione rivista in breve tempo, anche con il Vostro aiuto, grazie!

INDI CE

Prefazione

Il principale obiettivo di questo modulo e' l'introduzione alla programmazione con particolare riguardo al linguaggio C/C++. In tale ottica e' stato scelto di presentare un sottoinsieme dei costrutti del C++. In questo senso il linguaggio utilizzato può essere chiamato semplicemente $C+$.

Come e' noto il C++ e' un linguaggio per la programmazione ad oggetti e come tale permette l'utilizzo di svariate funzionalità tipiche dei modelli ad oggetti. Al fine di renderlo più facilmente raggiungibile dagli studenti dei primi anni del diploma in Ingegneria alcuni dei costrutti più complessi non vengono presentati. Alcuni di questi sono troppo complessi per un programmatore alle prime armi altri sono talmente potenti che possono portare il programmatore alle prime armi ad abusarne e quindi a produrre programmi difficilmente leggibili e manutenibili, avviandolo verso uno stile di programmazione non confacente con la produzione di software in ambito industriale.

P. Nesi
(l'autore)

1 Modulo: Le Strutture Informative

In questo modulo sono presentate le strutture informative e le operazioni che possono essere effettuate su tali strutture.

1.1 Le strutture Informative

Spesso le informazioni elaborate dai programmi non sono dati singoli ma appartengono a insiemi di dati che possono essere sia *omogenei* sia *eterogenei*.

Un insieme di dati e' *eterogeneo* quando sono in esso contenuti dati diverso tipo. Si consideri per esempio l'indirizzo di una persona questo e' composto da informazioni di vario tipo: numeri, nomi, etc. Un insieme di dati e' *omogeneo* quando contiene solo dati dello stesso tipo.

L'indirizzo stesso e' una struttura informativa, ma anche un'eventuale lista di indirizzi organizzati come una tabella e' una struttura informativa. Gli elementi della lista di indirizzi sono i singoli indirizzi mentre gli elementi dell'indirizzo sono le sue componenti.

Le strutture informative possono essere *astratte* o *fisiche*.

Le strutture *astratte* vengono utilizzate per descrivere le relazioni fra componenti in modo astratto definendo le proprietà di tali componenti. Le strutture *fisiche* descrivono come le componenti sono organizzate nella memoria del computer/elaboratore. La struttura fisica della memoria non e' sufficientemente flessibile per modellare tutte le strutture dati che sono necessarie ad un sistema informativo. Pertanto, si realizzano delle strutture fisiche suppletive che astraggono dalla struttura lineare della memoria.

Le strutture *fisiche* si dividono in strutture *interne* e *esterne*.

Le strutture interne descrivono l'organizzazione in memoria di base in modo da superare le limitazioni della linearità della memoria. Quelle esterne vengono definite allo stesso scopo ma sono quelle utilizzate per gestire i dati sulle unita' di memoria di massa e per superare i limiti di tali supporti.

Le strutture interne o esterne possono essere *statiche* o *dinamiche*.

Vengono dette strutture *statiche* quelle la cui organizzazione in termini di strutture fisiche e quindi di memoria di base viene decisa direttamente prima dell'esecuzione del programma. Le strutture statiche non possono cambiare le proprie dimensioni durante il loro utilizzo.
Sono dette strutture *dinamiche* quelle la cui realizzazione in termini di memoria di base viene effettuata solo al momento dell'esecuzione del programma. Le strutture dinamiche possono cambiare le proprie dimensioni durante il loro utilizzo, cioè durante l'esecuzione stessa del programma o della sua vita in genere.

Per esempio l'elenco telefonico e' una struttura dinamica poiché il numero degli elementi nell'elenco può cambiare nel corso di vari anni.

1.2 Operazioni sulle strutture

Sulle strutture si possono effettuare varie operazioni.

Tipicamente queste si dividono in operazioni *locali* o *globali*.

1.3 Operazioni locali

Sono operazioni *locali* quelle che interessano un solo elemento della struttura:

- la modifica di un elemento della struttura dati, modificando le componenti del singolo elemento.
- la cancellazione di un elemento della struttura dati preservando la struttura base.
- l'inserimento di un elemento nella struttura dati che passa ad avere un elemento in più.
- La visualizzazione delle componenti del singolo elemento della struttura dati.

1.4 Operazioni Globali

Sono operazioni *globali* quelle che interessano tutta la struttura:

- La concatenazione di due strutture dati compatibili per creare una struttura dati unica a partire da due o più strutture dati della stessa natura.
- La visita della struttura dati per: la stampa su carta o a video, per la ricerca (per esempio sulla base di una componente dell'elemento della struttura: la ricerca di un numero nella guida telefonica per cognome e nome).
- L'ordinamento degli elementi della struttura dati. Per esempio, l'ordinamento alfabetico rispetto ad una componente degli elementi delle componenti, oppure l'ordinamento per valore numerico della media o dei numeri di matricola in un archivio di studenti.
- La compattazione di una struttura dati che consiste nel rimuovere le componenti non più utili come quelle cancellate e quindi porta ad ottenere una struttura dati ridotta rispetto a quella iniziale.
- Il salvataggio su memoria di massa dell'intera struttura dati
- Il caricamento da memoria di massa dell'intera struttura dati.

1.5 Le strutture astratte

Le strutture astratte più importanti sono:

- la stringa,
- il vettore,
- la matrice,
- il record,
- la tabella,
- la lista,
- la pila,
- la coda,

- la coda circolare, // forse da escludere
- l'albero, // forse da escludere
- l'albero binario, // forse da escludere
- il grafo, // forse da escludere

1.6 La stringa

La stringa e' un insieme ordinato di simboli estratti da un certo alfabeto.

La stringa ha simboli ordinati nel senso che e' possibile riferirsi ai singoli simboli indicandoli per mezzo della loro posizione nella stringa partendo dal primo all'ultimo secondo un criterio di ordinamento.

Nel caso degli elaboratori l'alfabeto e' quello dei caratteri rappresentabili. Per esempio la tabella ASCII. Di seguito sono riportati alcuni esempi di stringhe:

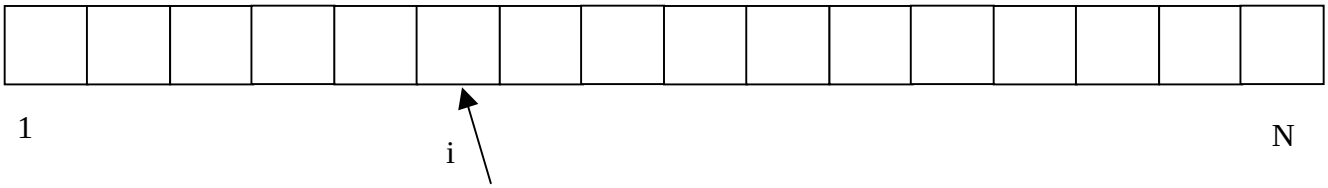
```
pippo
arciBaldo
asdfkljaslkjf
21421h34kl
347aAZ(*7jhewf12374*^223
:"{DÄ<DF**@#JHDF(AD
```

Ogni stringa ha una certa lunghezza pari al numeri di caratteri che essa contiene. La stringa che non contiene nessun carattere e' detta stringa vuota.

1.7 Il vettore

Il vettore e' un insieme ordinato di elementi di un certo tipo. Il vettore ha elementi ordinati nel senso che e' possibile riferirsi alle singole componenti indicandole per mezzo di un indice (per esempio l'indice `i` nella seguente figura). Tale indice identifica in modo ordinato gli elementi del vettore dal primo all'ultimo, secondo un criterio di ordinamento, per esempio, crescente.

Il vettore ha una dimensione pari al numero di componenti/elementi di cui e' composto, la dimensione e' fissa, nel senso che non può tipicamente essere variata dopo la sua realizzazione. I vettori sono tipicamente strutture statiche. Gli elementi sono tutti dello stesso tipo, pertanto il vettore e' un struttura dati omogenea. Per esempio, si possono avere vettori di numeri interi, naturali, reali, Booleani, etc.



Il concetto di vettore e' di fondamentale importanza applicativa pressoché in ogni disciplina scientifica. L'idea di aggregare informazioni omogenee in un'unica entità e di poterle successivamente reperire in virtù della loro posizione conduce spesso a formulare problemi

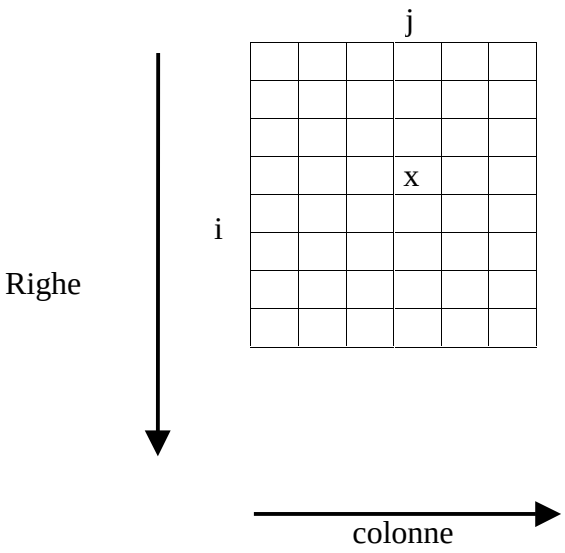
in maniera estremamente compatta ed elegante. Le operazioni dell'algebra lineare offrono poi un supporto insostituibile per una soluzione efficiente e spesso riconducibile a schemi classici. Tuttavia, il concetto di vettore considerato nell'ambito dell'informatica non si limita ad ospitare il concetto algebrico. Il vettore risulta una struttura ideale per rappresentare tabelle contenenti informazioni, soprattutto qualora esse siano di natura essenzialmente statica, ma anche per rappresentare funzioni mediante la memorizzazione dei valori del codominio.

Un vettore V avente dimensione N viene rappresentato indicando i suoi elementi con V(i) oppure con Vi, ma viene anche considerato nella sua interezza semplicemente con V.
Vi o V(i) rappresentano l'elemento i-esimo del vettore V, come V(4) e' il quarto elemento del vettore se il vettore ha come indice i che va da 1 a N, altrimenti e' il quinto elemento se il vettore ha come indice i che va da 0 a N-1.

```
V = { Vi | i = 1 … N},     oppure    V = { Vi | i = 0 … N-1}
```

1.8 La Matrice

La matrice e' la naturale estensione bidimensionale del vettore. La matrice ha elementi ordinati nel senso che e' possibile riferirsi alle singole componenti indicandole per mezzo di due indici partendo dal primo all'ultimo, secondo un criterio di ordinamento che tiene conto di entrambe le dimensioni. Si consideri l'elemento A(i,j) della matrice A, i indica la riga partendo dall'alto con la riga 0 o 1, j indica la colonna partendo da sinistra con la colonna 0 o 1.
Le matrici devono essere composte da elementi dello stesso tipo, sono pertanto strutture omogenee. Le matrici, come i vettori sono tipicamente strutture statiche nel senso che non e' possibile cambiare le dimensioni della matrice dopo la sua creazione iniziale.
La matrice puo' avere un numero di colonne diverso da quello delle righe, nel qual caso si chiama matrice rettangolare. Si chiamano matrici quadrate quelle che hanno un egual numero di righe e colonne.



La posizione dell'elemento di coordinate (0,0) oppure (1,1), cioe' del primo elemento della matrice dipende dalla convezione utilizzata. Tipicamente e' in alto a sinistra (come nell'esempio) oppure in basso a sinitra.

1.9 Il record e i tipi

Un record e' un insieme ordinato ed eterogeneo di elementi. Il singolo elemento del record può essere chiamato anche campo o componente. In un record si possono avere componenti di vario tipo: interi, naturali, stringhe, vettori di interi, etc. Il record e' tipicamente una struttura statica nel senso che non e' possibile variarne il numero di componenti dopo la sua creazione iniziale. Di seguito viene riportato un esempio di record:

```
cognome = Rossi
nome = Mario
via = Via dello Steccuto
numero = 45
```

Ogni campo del record viene identificato dal nome del campo. `Cognome` e' il nome del primo campo, `Rossi` e' il contenuto del primo campo del record. Il campo `Cognome` e' di tipo stringa mentre il campo `numero civico` e' un numero intero. La rappresentazione del record può essere effettuata nel seguente modo:

```
{ cognome, nome, via, numero }
```

Tipicamente il record stesso o meglio la struttura che lo definisce ha un nome. Spesso il concetto di record e' confuso con quello di tipo. Un tipo può avere al suo interno un record per rappresentare i suoi dati ma in realtà ciò che caratterizza un tipo e' l'insieme delle operazioni che possono servire per manipolarlo.

1.10 La tabella

Le tabelle sono una generalizzazione del concetto di matrice. La tabella a differenza della matrice, puo' contenere dati di tipo diverso. Per esempio numeri, commenti, descrizioni, etc. Tipicamente, ad ogni colonna della tabella e' associato un tipo e quindi un significato.
La tabella e' un insieme ordinato ed omogeneo di record. Tipicamente la tabella e' una struttura dati statica nel senso che le sue dimensioni non variano dopo la sua realizzazione iniziale.

Un record della tabella può essere identificato in due modi: per indice posizionale, o per chiave. L'indice posizionale non e' altro che l'indice che identifica il singolo record partendo dal primo fino all'ultimo. In questo senso la tabella si può vedere come un vettore di record.

Nome	Cognome	Età	Telefono	Indice
Carlo	Zolli	14	0123413412	1
Pino	Polli	24	0234242444	2
Gino	Arbi	4	0345543535	3
Lino	Corbi	56	0455436266	4
….	….	….	….	…

1.11 Tabella con accesso per chiave

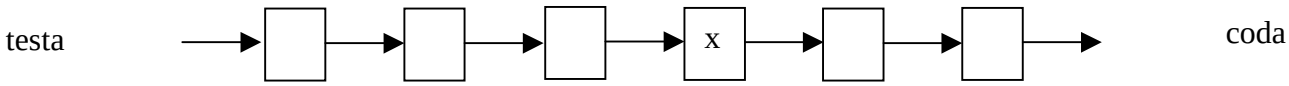
Per chiave significa che il singolo record viene identificato per mezzo del contenuto stesso di un campo del record. Per esempio nella tabella che segue l'identificazione del record potrebbe essere effettuata per numero di telefono, per età, per cognome, etc. L'individuazione di un record per chiave significa identificare la posizione del record nella tabella (il record di Gino e' in posizione di indice 3, se l'indice di ordinamento va da 1 a N).

La chiave per la quale si effettua una ricerca al fine di individuare il singolo record dovrebbe essere sufficientemente significativa da identificare in modo univoco il record. Questo puo' non essere possibile, per esempio nella ricerca per cognome vi possono essere molti record con lo stesso cognome. Per risolvere tale problema si possono assegnare delle chiavi univoche oppure comporre i campi dei record per definire delle chiavi piu' significative. Per esempio nome piu' cognome.

Nome	Cognome	Età	Telefono
Carlo	Zolli	14	0123413412
Pino	Polli	24	0234242444
Gino	Arbi	4	0345543535
Lino	Corbi	56	0455436266
….	….	….	….

1.12 La lista

Le liste sono strutture dati che possono essere omogenee e non omogenee, lineari e non lineari.
Sono dette liste lineari quelle organizzate come un insieme ordinato di elementi. Ogni elemento ha un successivo e un precedente ad esclusione del primo che non ha precedente e dell'ultimo che non ha successivo.



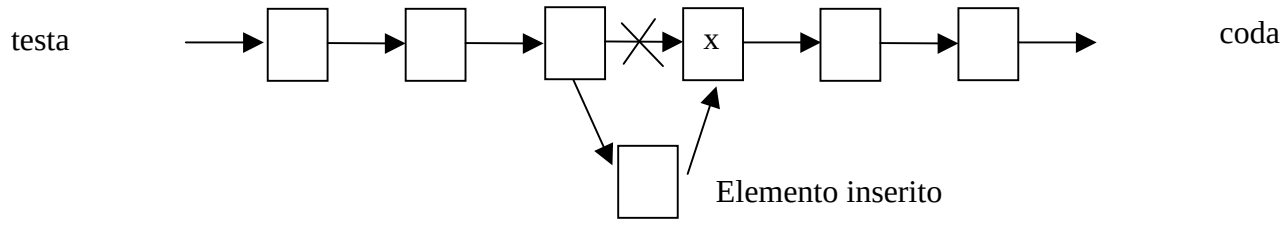
Questo significa che x.succ() indica l'elemento successivo all'elemento x. x.prec() indica l'elemento precedente a x. In questa struttura, per accedere all'elemento finale e' necessario partire dal primo elemento della lista (detto "la testa", l'elemento a sinistra nel disegno) e scorrere tutti gli elementi successivi fino all'ultimo (detto "la coda", l'ultimo elemento a destra nella figura). Questo tipo di organizzazione per l'accesso alle componenti viene detto sequenziale.

Le liste sono strutture dati ad accesso sequenziale. La lista può contenere tipicamente un numero variabile di elementi nel tempo, nel senso che la sua dimensione in termini di elementi può variare dopo la sua creazione iniziale, pertanto deve essere considerata una struttura dinamica. Alla sua creazione la lista e' tipicamente vuota, in seguito vengono inseriti gli elementi connettendoli secondo regole di ordinamento o di convenienza.

Su una lista si possono effettuare operazioni di inserimento, cancellazione, visita, modifica, creazione, concatenazione di due liste, etc. Una lista si crea inserendo il primo elemento.

1.13 Inserimento in una lista

L'inserimento in testa ed in coda è semplice da comprendere: si aggiunge un ulteriore elemento e lo si collega al resto della lista.
L'inserimento di un elemento all'interno della lista deve essere effettuato aprendo la lista ed ristabilendo i legami fra gli elementi:

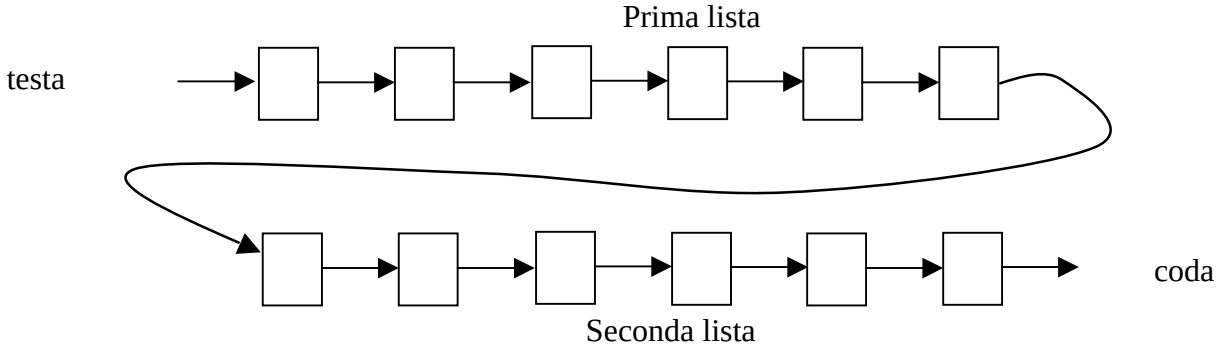


Si possono inserire anche elementi in testa ed in coda con modalità leggermente diverse come vedremo meglio in seguito.

La lista può essere vuota se non vi sono elementi. Una lista con un solo elemento può diventare vuota se questo viene cancellato.

1.14 Concatenazione di liste

Due liste possono essere concatenate nel senso che una lista può essere congiunta con un'altra in modo da formare un'unica lista con tutti gli elementi. Questo viene effettuato collegando la coda di una alla testa dell'altra:



1.15 Le differenze fra lista e vettore

Le maggiori differenze fra vettore e lista sono:

- Il vettore ha un accesso diretto (casuale) ai singoli elementi mentre la lista ha un accesso sequenziale.
- La lista può avere un numero variabile di elementi nel tempo, e' pertanto una struttura dinamica.
- La lista può anche contenere elementi di diverso tipo
- La lista può essere anche non lineare

1.16 Descrizione di liste lineari e non

Gli elementi di una lista B possono essere elencati nel seguente modo:

B (3, 5, 6, 77, 89)

In questo caso la lista ha 5 elementi. La lista e' una struttura ordinata che può contenere insiemi di dati anche di diverso tipo.

Sono possibili sulla lista le operazioni di ricerca, cancellazione, concatenazione, visita etc. che sono state elencate in precedenza. Tipicamente le operazioni di inserimento e cancellazione possono essere effettuate agli estremi della lista come in ogni punto della lista.

Si dicono liste non lineari quelle liste che contengono fra i loro elementi delle liste. Per esempio si noti la seguente lista G non lineare:

G (A, B, C, (d, F, r, g), G, (G, T))

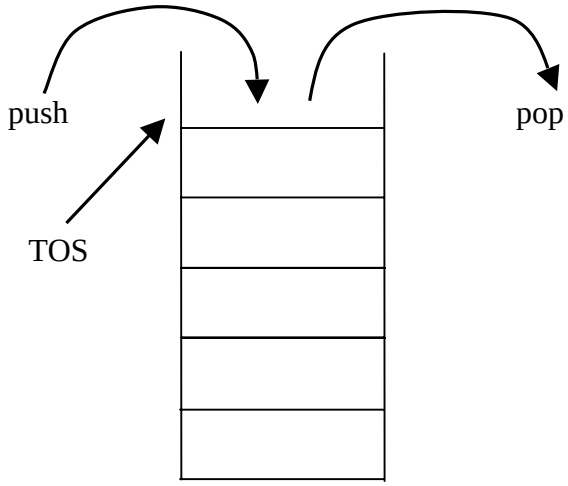
Questa lista contiene alcuni elementi singoli ma anche delle liste. Sono liste non lineari i grafi e gli alberi.

1.17 La pila (lo stack)

La pila o *stack* e' un struttura lineare dove le operazioni di inserimento ed estrazione dei dati avvengono sempre dalla stessa parte. La pila e' un struttura che e' governata da una politica di tipo FILO (*first input last output*) il primo a entrare e' l'ultimo a uscire, oppure equivalentemente LIFO (*last input first output*), l'ultimo a entrare e' il primo ad uscire. Si prenda ad esempio un tubetto di pasticche di vitamina. Per togliere l'ultima (la prima pasticca che e' stata inserita) e' necessario togliere tutte le pasticche presenti.

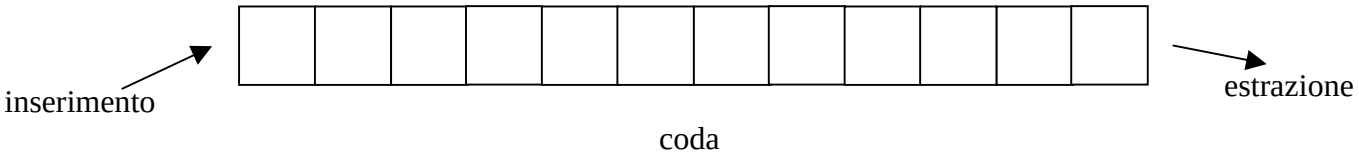
L'operazione di inserimento viene detta di *push* mentre l'operazione di estrazione viene detta di *pop*. Per la sua gestione e' necessario conoscere solamente il punto in cui viene fatta l'inserzione oppure l'estrazione. Tale punto e' detto *Top of the Stack*, TOS. Lo *stack* e' tipicamente una struttura dinamica perché le sue dimensioni possono non essere note al momento della realizzazione. Poiché tutte le operazioni fanno riferimento ad uno stato iniziale dello *stack*, stato definito alla sua realizzazione, e' necessario tenere sotto controllo quando lo *stack* si svuota. Durante la gestione dello *stack* può altrimenti

accadere che si cerchi di effettuare un'operazione di pop senza che vi sia un elemento da estrarre dallo *stack*. Pertanto le operazioni di base sono la creazione, l'inserimento (*pop*), l'estrazione (*push*) e la verifica di *stack* vuoto prima di fare ogni *pop*.



1.18 La coda (il buffer)

La coda o *buffer* e' una struttura lineare dove le operazioni di inserimento vengono effettuate da una parte mentre quelle di estrazione vengono effettuate dall'altra. Tipicamente la parte dove si fanno le estrazioni viene chiamata testa mentre quella ove si effettuano gli inserimenti viene chiamata coda, cioè la fine della struttura (si dice infatti "mettiti in coda"). La coda e' tipicamente una struttura dinamica nel senso che le sue dimensioni possono non essere note al momento della sua realizzazione. La creazione di una coda si opera inserendo il primo elemento, mentre la coda si svuota togliendo l'ultimo elemento rimasto. Pertanto le operazioni di base sono la creazione, l'inserimento, l'estrazione, la verifica di coda vuota e lo svuotamento.



coda

Il meccanismo di gestione della coda si basa sulla politica FIFO (*first input first output*), il primo ad entrare e' il primo ad essere servito, cioè ad uscire. Questo può essere espresso anche con la politica LILO (*last input last output*), l'ultimo ad entrare e' l'ultimo ad uscire. Tale meccanismo e' consistente a quello che dovrebbe accadere nelle code che siamo costretti ad effettuare nella vita quotidiana.

1.19 L'albero

Un albero ……..

1.20 L'albero binario

1.21 Il grafo

1.22 Le strutture Interne

Le strutture interne alla memoria sono:

- Il vettore di memoria
- La lista concatenata semplice
- La Lista concatenata doppia // forse da escludere

1.23 Il vettore di memoria, il puntatore

Il vettore di memoria e' una struttura fisica interna. Questo non deve essere confuso con il vettore come struttura astratta. Il vettore di memoria e' la struttura di memorizzazione più semplice, ed e' costituita da elementi memorizzati in celle contigue di memoria identificate da un indirizzo di memoria iniziando da un indirizzo di partenza (o di base) B che può essere zero, 0. L'indirizzo della memoria viene detto puntatore. Il *puntatore* non e' altro che l'indice all'interno del vettore di memoria che rappresenta la memoria stessa dell'elaboratore.

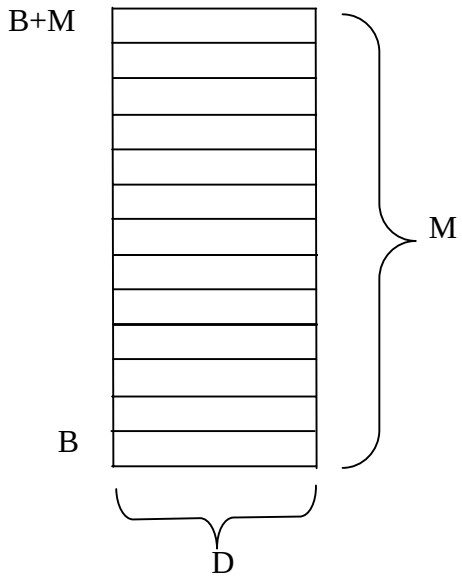
La memoria ha una dimensione M che viene espressa in numero di celle. Ogni cella può avere una dimensione D, da 1 a più byte: 2, 4, 8. Pertanto se si considera un insieme di numeri elementi interi di 2 byte che sono inseriti nel vettore di memoria a partire dal suo inizio, l'elemento i-esimo del vettore si troverà ad avere il seguente indirizzo di cella:

```
indirizzo di cella = B + 2 x i
```

L'effettiva dimensione in byte della memoria viene stimata considerando la dimensione della cella ed il numero delle celle: M x D. L'indirizzo di cella non deve mai superare tale valore aumentato dell'indirizzo di partenza B:

```
indirizzo di cella < = B + M x D
```

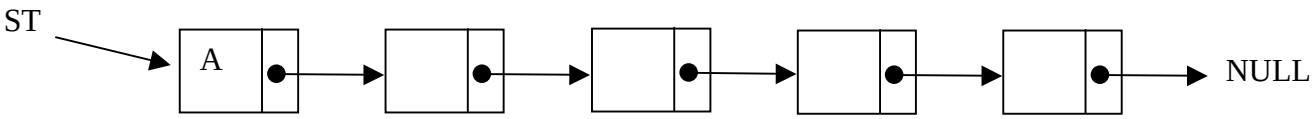
Il vettore di memoria può essere utilizzato per contenere tutte le strutture astratte che sono state discusse in precedenza. Il vettore di memoria ha una struttura rigida e pertanto la loro realizzazione risulta semplificata per la presenza del vettore di memoria come struttura fisica, per altre la gestione risulta complessa ma sempre possibile.



1.24 La lista concatenata semplice, catena semplice

La lista concatenata o catena semplice e' una struttura interna ordinata per la quale elementi consecutivi non sono disposti in memoria fisica in celle consecutive. La sequenza e quindi l'ordine degli elementi della lista viene dato in base a dei legami che vengono definiti fra gli elementi stessi della lista. I singoli elementi della lista possono essere dei record. Ogni record contiene un contenuto informativo, un campo chiave che distingue il record rispetto agli altri e un campo che contiene il riferimento all'elemento successivo. Il campo informativo può essere mancante.

Ogni elemento della lista ha un riferimento (puntatore/indirizzo di memoria) all'elemento successivo. Il primo elemento viene identificato da un puntatore, nell'esempio ST. L'ultimo elemento non riferisce nessun elemento e pertanto il suo puntatore viene imposto a un valore prefissato nullo, NULL.



Questa struttura dati e' molto più flessibile rispetto al vettore poiché e' possibile inserire elementi anche all'interno della lista (senza dover riorganizzare la lista) e non solo alle estremità come nella coda. Ogni elemento della lista contiene una parte dati e una parte per referenziare l'elemento successivo. Questo viene tipicamente effettuato tramite un puntatore di memoria e quindi lo spazio in termini di byte per tale componente del singolo record deve essere adeguato.

Se si conosce un elemento, per esempio l'elemento A dell'esempio, si può fare riferimento all'elemento successivo nella lista (seguendo l'ordine del puntatore) tramite la notazione A.succ().

La lista e' una struttura dati sequenziale pertanto non e' possibile raggiungere il singolo elemento se non si conosce il suo indirizzo di memoria tramite un puntatore.

A parità di numero di elementi, questo tipo di struttura dati necessita, rispetto al vettore di memoria, di un maggior numero di byte di memoria per la presenza dei puntatori agli elementi successivi. Tale struttura dati e' pero' più flessibile poiché le seguenti operazioni sono piu' semplici che nel vettore.

- La cancellazione implica semplicemente la cancellazione dell'elemento e l'aggiornamento del puntatore.

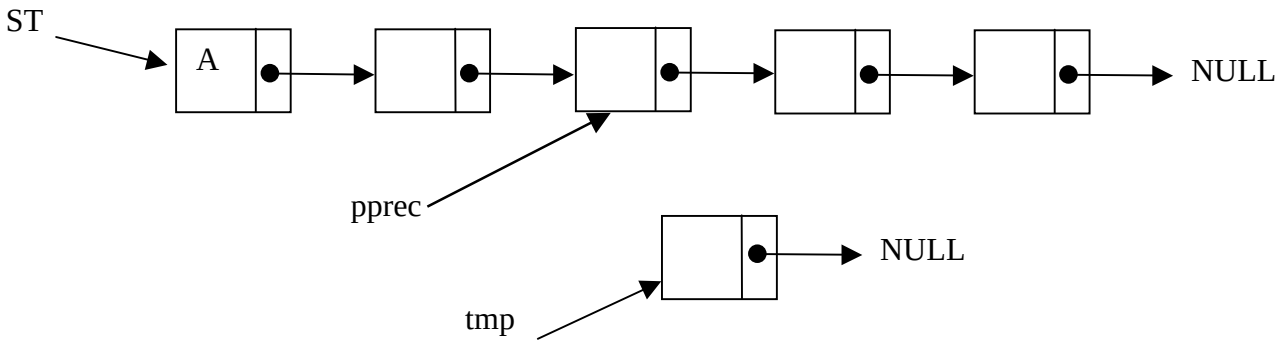
- L'inserimento implica semplicemente l'inserimento di un nuovo elemento riorganizzando i puntatori ove questo e' inserito. Le operazioni di inserimento in testa ed in coda sono semplici, si possono complicare a causa della mancanza di informazione relativa alla fine della lista.

1.25 Inserimento in lista concatenata semplice, catena semplice

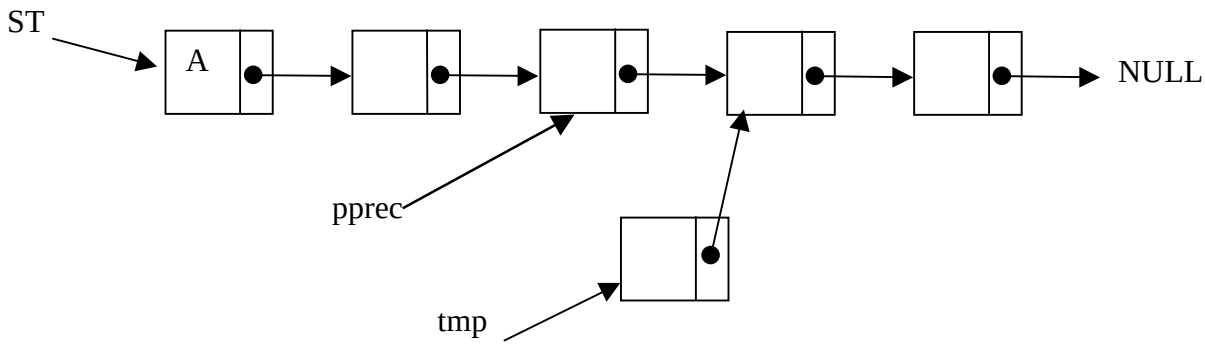
L'inserimento di un elemento lungo la lista e' un'operazione piu' complessa. A tal fine e' necessario interrompere la catena per aggiungere un ulteriore elemento ristabilendo i legami. In questo procedimento si deve fare attenzione a non perdere i riferimenti ai due tronconi della lista, altrimenti si rischia di perdere un parte della lista.

Si procede in tre fasi.

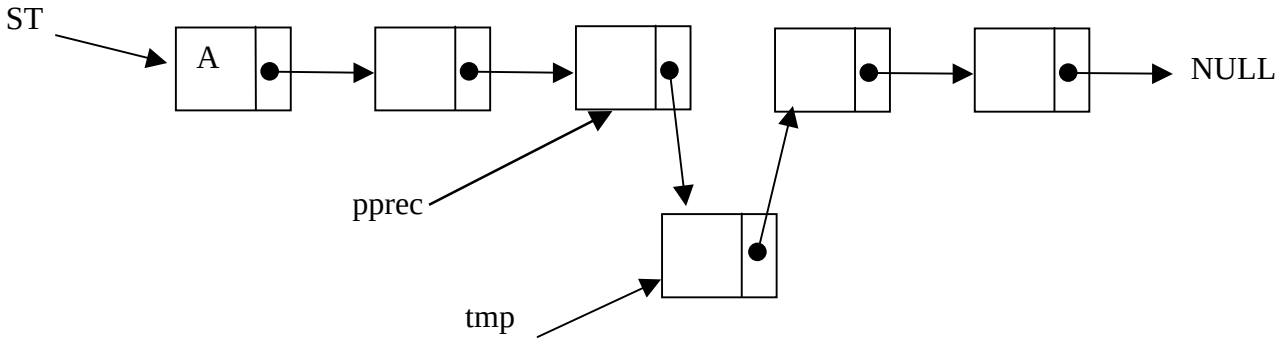
Prima fase: si ha sia la lista che il nuovo elemento da inserire. Questo e' praticamente una lista di un solo elemento il puntatore `tmp` tiene conto della sua posizione in memoria. E' necessario avere un riferimento/puntatore all'elemento successivamente al quale si vuole effettuare l'inserimento, `pprec` nella figura.



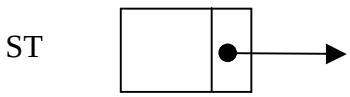
Seconda fase: l'elemento da inserire viene congiunto alla seconda parte della lista



Terza fase: l'elemento precedente viene congiunto all'elemento da inserire assegnando il valore di `tmp` a valore del puntatore al successivo di `pprec`. A questo punto l'operazione e' conclusa e i puntatori `pprec` e `tmp` non sono piu' necessari.



1.26 La Lista concatenata doppia



1.27 Realizzazione di Strutture astratte su strutture fisiche

Le strutture astratte possono essere implementate utilizzando delle strutture fisiche. Gli elaboratori tipicamente sono dotati di una memoria di massa che segue un'organizzazione vettoriale. Pertanto le varie combinazione fra strutture astratte e fisiche determinano possibili realizzazioni. Nella tabella seguente viene data una valutazione della difficoltà della realizzazione di strutture astratte su strutture interne.

	Vettore	**Lista**
Stringa	bassa	Medio-bassa
Vettore	bassa	Medio-bassa
Record	Media	Media
Tabella	Bassa	Media
Lista	Media	Bassa
Pila/Stack	Media	Bassa
Coda/Buffer	Media	Bassa

Da questa tabella si evince che le strutture che necessitano di un maggior dinamismo in termini di inserzione e estrazione di elementi possono essere implementate con maggiore facilita' in strutture flessibili come la lista che con strutture come il vettore.

2 Modulo: Variabili, operazioni e costanti

In questo modulo sono presentati i concetti di variabile e costante. Sono inoltre introdotto il concetto di tipo e i tipi elementari. Viene quindi esposto il concetto di dichiarazione sia di variabile sia di costanti includendo anche la loro inizializzazione.
Su tale base viene proposta l'istruzione di assegnazione e le varie operazioni che si possono effettuare su variabili e costanti all'interno di espressioni. In questo contesto sono considerate le operazioni su interi, reali, Booleani e caratteri. Per le operazioni sono introdotte le precedenze fra questi e la loro semantica.
Gli operatori C++ piu' complessi per la manipolazione delle strutture sono lasciati alla parte sulle strutture dati.
Sono presentati anche gli operatori di confronto e le istruzioni di incremento tipiche del C/C++.
Infine vengono proposti i tipi enumerativi, le assegnazioni con operatore del C/C++.

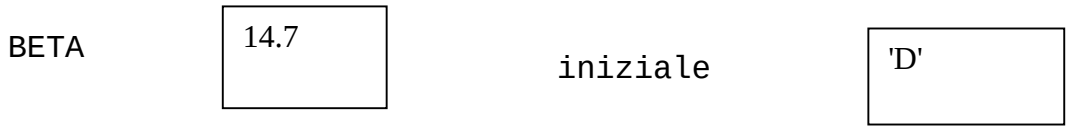
Particolare riguardo e' stato dato al controllo di consistenza fra tipi e alle conversioni di tipo.

2.1 Variabili e costanti

Ogni programma lavora su un certo numero di dati. Questi dati possono assumere un valore costante per tutto lo svolgimento dell'algoritmo: l'esecuzione del programma. In tal caso vengono detti *costanti.*

Nella stesura di programmi vi e' la necessita' di utilizzare valori temporanei di dati. Questi valori possono essere memorizzati in celle di memoria che possono essere viste come scatole contenenti informazioni. Le

informazioni contenute in tali scatole possono essere modificate durante l'esecuzione del programma. Pertanto, tali scatole vengono comunemente chiamate *variabili.*
Le variabili e le costanti possono assumere informazioni di diverso tipo come, valori numerici, caratteri, puntatori, e informazioni più complesse.
Durante la stesura di un programma si può fare riferimento a variabili e/o a costanti per mezzo di nomi alfanumerici. Nel seguente esempio sono riportati due quadrati. Uno di nome `BETA` che contiene il valore 14.7 e l'altro di nome `iniziale` che contiene il carattere ASCII 'D'.
Nel precedente esempio non e' stata fatta alcuna distinzione fra `BETA` e `iniziale` riguardo alla loro natura di costanti o variabili.



Nella stesura di un programma e' corretto utilizzare costanti quando queste riportano dati che non cambiano durante l'esecuzione del programma, mentre e' corretto utilizzare variabili quando e' necessario cambiarne il valore.

2.2 Nomi delle variabili

La struttura lessicale dei nomi che possono essere dati alle variabili e alle costanti dipende dal linguaggio che si utilizza per la codifica del programma. In C/C++ come in altri linguaggi di programmazione i nomi non devono iniziare con un numero e non possono contenere spazi. Possono contenere lettere minuscole e maiuscole e numeri.
Esempi di nomi validi sono: `Der56, der56, kio666ooo, albero, senza_parole, sono_una_variabile.`
Esempi di nomi non validi sono: `45der, sono una variabile, Beta 3, bg-gt.`

Nella prima lista di nomi e' stato utilizzato il simbolo detto *underscore*: '_' per sostituire lo spazio. Nella seconda lista di nomi non validi: il primo non e' valido per la presenza di un numero come primo carattere, mentre il secondo e il terzo non sono validi per la presenza di spazi, il quarto e' incorretto per l'uso del simbolo '-' che ha significato matematico e non di separatore di parti di un nome.

In C/C++ come in altri linguaggi, i nomi di variabili e costanti sono sensibili al tipo di caratteri che si utilizzano, cioè sono come si suole dire *key-sensitive*. Questo significa che in tali linguaggi due dati con nomi `Der56, der56, DER56, eRr56`, fanno riferimento a variabili o costanti diverse.
In seguito, al fine di rendere più semplice la lettura e la comprensione degli esercizi saranno utilizzati nomi composti solo da caratteri maiuscoli e numeri per le costanti. Le variabili avranno nomi anche misti ma con alcune lettere minuscole. In questo modo sarà immediato distinguere fra variabili e costanti.

2.3 Tipi di dato, rappresentazione e operazioni

Variabili e costanti possono rappresentare in un programma informazioni di vario tipo: numeri, caratteri, informazioni strutturate.

Nella parte relativa alla rappresentazione dei dati sono stati utilizzati alcuni tipi di dati di uso comune: numeri interi, numeri in virgola mobile, caratteri, variabili *Booleane*. Quando un certo tipo di dato viene utilizzato nella stesura di un programma questo assume una specifica *rappresentazione* e quindi un certo *ingombro in memoria*, una certa dinamica e precisione.

In informatica, il *tipo* di dato determina l'insieme dei valori che tale dato può assumere e le operazioni che possono essere effettuate per manipolarlo, cioè il dominio del tipo di dato.
I valori che un tipo di dato può assumere dipendono dalla natura del tipo di dato (intero, naturale, etc.) ma anche dalla sua rappresentazione in termini di bit.
Il numero e la semantica delle operazioni che possono essere svolte su un certo tipo di dato dipendono dalla natura del tipo di dato e dalla loro realizzazione per lo specifico linguaggio di programmazione che viene utilizzato. Operazioni simbolicamente uguali possono dare avere una diversa semantica in base al tipo di dato sul quale sono effettuate. Si pensi alla somma di numeri: (i) interi in rappresentazione forma complemento, (ii) in virgola mobile; (iii) complessi.

2.4 Tipi elementari o composti

I tipi possono essere elementari o strutturati. I tipi elementari sono quelli disponibili direttamente dal linguaggio, cioè i tipi predefiniti, di base o elementari. I tipi strutturati possono essere definiti per composizione di tipi elementari. Questi vengono tipicamente definiti dai programmatori. Esempi di tipi strutturati sono la data, il codice fiscale, l'indirizzo. Per la definizione di un nuovo tipo rivestono particolare importanza la definizione e la realizzazione delle operazioni consentite sulle variabili e costanti del nuovo tipo di dato. La maggiore parte dei linguaggi di vecchia generazione (come FORTRAN, COBOL, BASIC) non permettono la definizione e realizzazione di nuovi tipi di dati mentre i linguaggi più moderni come PASCAL e C, permettono la definizione di nuovi "tipi" di dati senza però permettere la definizione e realizzazione di nuovi operatori. Infine, i linguaggi orientati agli oggetti basano la loro potenza proprio su questi aspetti.
In seguito faremo riferimento alla potenzialità del C++ per la realizzazione di tipi di dati.

Oltre ai tipi elementari e strutturati vi sono anche i puntatori che rappresentano indirizzi di memoria di variabili. Pertanto si possono avere puntatori che referenziano qualsiasi tipo di dato: elementare, strutturato o puntatore.

Variabili e costanti per contenere tali tipi di informazioni possono essere utilizzate all'interno di programmi. Il loro utilizzo dipende dalle operazioni che possono essere eseguite su ogni tipo di dato e dalla loro rappresentazione in memoria. Questi fattori dipendono dal linguaggio di programmazione e in alcuni casi anche dal Sistema Operativo. In seguito sarà fatto specifico riferimento al C/C++.

2.5 I tipi elementari del C/C++

I principali tipi di dati che sono disponibili come predefiniti secondo lo standard ANSI del C sono riportati nella seguente tabella. Per ogni variabile e' riportato il nome utilizzato (la *keyword*) nella realizzazione di programmi, l'ingombro in memoria in byte, la dinamica e una breve descrizione. La Dinamica riporta il minimo ed il massimo valore che può essere assunto da quel tipo di variabile.

Tipo	Byte	Dinamica	Descrizione
`int`	2	-32767, +32767	Numero interi. Rappresentazione in forma complemento a 2.
`unsigned int`	2	0-65536	Numeri naturali, rappresentazione binaria.
`long`	4	$\pm 2^{31}$	Rappresentazione in forma complemento a 2
`unsigned long`	4	2^{32}	Numeri naturali, rappresentazione binaria.
`char`	1	0-255	Codifica ASCII dei caratteri
`unsigned char`	1	0-255	Numero naturale o carattere.
`float`	4	$\pm 3.4 \cdot 10^{38}$	Numero reale. Rappresentazione in virgola mobile.
`double`	8	$\pm 1.80 \cdot 10^{308}$	Numero reale. Rappresentazione in virgola mobile.

In C/C++, si possono definire puntatori a variabili di tipo elementare e/o definito dal programmatore come tipo strutturato, ma anche puntatori a puntatori..
In seguito saranno discussi i meccanismi per la definizione e realizzazione di tipi strutturati e per la gestione di puntatori.

2.6 Dichiarazioni di variabili

Nella maggior parte dei linguaggi di programmazione, variabili e costanti devono essere dichiarate prima del loro utilizzo in modo esplicito. La dichiarazione e' una proposizione in cui le variabili e le costanti che saranno utilizzate nel programma vengono elencate dichiarandone il loro tipo. In altri linguaggi la dichiarazione di certe variabili e/o costanti e' implicita', oppure viene fatta in base a delle restrizioni lessicali sul nome stesso. Per esempio in BASIC e FORTRAN il primo carattere può distingue la variabile come carattere o numero intero.

In C/C++ le variabili possono essere dichiarate nel seguente modo:

```
int a; // dichiarazione della variabile intera a
float c, b; // dichiarazione delle variabili float c, b
```

Il tipo della variabile deve essere seguito da uno o più nomi delle variabili, intervallati da virgole.
L'esempio precedente riporta una porzione di programma in cui sono state dichiarate tre variabili distinte di due tipi diversi. L'istruzione di dichiarazione viene chiusa con un punto e virgola.
Il testo riportato a destra della dichiarazione a partire dalle due barre "//" fino alla fine del rigo costituisce un commento che può essere incluso nel programma ma non viene preso in considerazione come istruzione di programmazione dal compilatore.

2.7 L'assegnazione

I valori delle variabili possono essere modificati tramite il costrutto di assegnazione. Con l'assegnazione e' possibile copiare in una variabile i valori calcolati tramite delle espressioni algebriche o contenuti in altre variabili o costanti. L'assegnazione si opera in C/C++ con la seguente formalizzazione BNF:

```
<variabile> = <espressione>;
```

Dove per espressione si intende una espressione algebrica comunque complessa definita in base ad una combinazione di operandi (variabili, costanti) ed operatori (algebrici, Booleani, di confronto) come sarà mostrato in seguito.
L'operazione di assegnazione richiede di calcolare l'`<espressione>` sulla destra del segno di uguale per poi assegnare il valore alla variabile posta sulla sua sinistra. Per poter procedere con l'assegnazione il valore prodotto dall'espressione deve essere compatibile con il tipo della variabile. Per esempio se la variabile e' di tipo intero allora anche l'espressione deve produrre un risultato di tipo intero.

Per esempio con il seguente programma:

```
int a, b;
a = 15;
b = 15 + 1;
```

vengono dichiarate le variabili `a` e `b` di tipo `int,` alla prima viene assegnato il valore di 15, alla seconda il valore di 16. L'uguale rappresenta il simbolo dell'assegnazione, questo non deve essere confuso con il simbolo rappresentato da due = consecutivi ==, che rappresenta un confronto per uguale come discusso in seguito.

L'operazione di assegnazione deve essere definita per poter essere utilizzata fra tipi di dati. Nel precedente esempio e' corretto assegnare 15 alla variabile `a` poiché questa e' intera. Non e' altrettanto corretto, per esempio, assegnare un valore reale ad una variabile intera.

E' pertanto necessario parlare di meccanismi di conversione fra tipi di dati e della struttura delle espressioni che si possono scrivere in base alle variabili dichiarate. Prima di passare a questi aspetti ci preme mostrare come sia possibile realizzare il primo programma.
A questo fine saranno introdotti altri concetti semplici anche in modo non completo per dare la possibilità di capire fin da subito il concetto di programma.

2.8 Inizializzazione di variabili elementari

Il valore delle variabili può essere imposto alla loro dichiarazione tramite un meccanismo di inizializzazione. Tale meccanismo si basa sull'assegnazione di un valore direttamente al momento della dichiarazione. In C/C++ si effettua nel modo seguente utilizzando l'operatore di assegnazione:

```
int a=9;
```

In questo caso, alla variabile a e' stato assegnato il valore 9 durante la sua dichiarazione. L'operazione di inizializzazione segue le regole dell'assegnazione, nel senso che il valore assegnato deve essere dello stesso tipo della variabile per non creare problemi di conversione. I problemi di conversione saranno trattati nella sessione 2.24.
I numeri interi possono essere impostati ed inizializzati anche utilizzando codifica esadecimale o ottale. Per esempio:

```
int b = 0x45;   // inizializzazione con notazione esadecimale
int a = 0145;   // inizializzazione con notazione ottale
```

I numeri in notazione esadecimale possono essere facilmente riconosciuti ed impostati utilizzando il prefisso 0x, mentre per inserire un numero ottale e' sufficiente avere uno 0 (zero) come primo carattere del numero.

2.9 Scambio di Valore fra due variabili

In questa sessione viene proposto un semplice esercizio di programmazione. Il problema consiste nello scambio del valore di due variabili. Si ipotizzi di avere due variabili con due distinti valori. Per esempio:

```
int a= 5, b=6, c;
```

L'obiettivo del seguente programma e' di scambiare il valore di tali variabili. A questo fine si utilizza una terza variabile di appoggio c e si procede come segue:

```
// a==5, b==6
c = a;
// c==5, a==5, b==6
a = b;
// c==5, a==6, b==6
b = c;
// c==5, a==6, b==5
```

Dopo ogni passo sono stati riportati i valori delle variabili in questione. E' chiaro da questo semplice programma che per effettuare lo scambio e' stato copiato il valore della variabile a nella variabile temporanea c, quindi si e' proceduto a copiare il valore della variabile b in a e alla fine il valore della variabile c (copia temporanea del valore di a) in b. In questo modo le due variabili si trovano ad avere i valori scambiati.

2.10 Operazioni fra variabili Intere

La semantica e il numero delle operazioni aritmetiche consentite per ogni tipo di dato dipendono dal compilatore. Per le variabili intere (int e long) sono tipicamente disponibili le operazioni aritmetiche di somma, prodotto, divisione e sottrazione con gli operatori: +, *, /, -.

La somma e la sottrazione sono regolate dai meccanismi della rappresentazione complemento a 2. L'operazione di prodotto può dare luogo facilmente a traboccamenti. L'operazione di divisione provoca anche il troncamento del risultato:

```
int a = 15,b = 2, c;
c = a/b;
```

Il valore assunto da `c` e' di 7, ottenuto troncando 7.5.

Fra numeri interi e' consentita **l'operazione di modulo**. Tale operazione viene specificata con l'operatore binario % che calcola il resto della divisione.

```
int a = 15, b = 2, c;
c = a % b;
```

Il valore assegnato alla variabile `c` e' pari a 1, il resto di 7*2 rispetto a 15. Se il dividendo (primo operando dell'operatore di modulo) e' negativo il risultato sarà pari al resto con segno negativo. Il segno del divisore non influenza il risultato. Questo comportamento differenzia il C dal Pascal e da altri linguaggi di programmazione. Riguardo a questo operatore in queste condizioni si possono avere comportamenti diversi indipendentemente dal compilatore che viene utilizzato anche per lo stesso linguaggio. Quest'operazione non e' consentita fra variabili in virgola mobile.

In tutte le operazioni che sono state mostrate per i numeri interi possono figurare costanti in notazione decimale, esadecimale o ottale.

I numeri interi di tipo long espressi nella forma decimale dovrebbero essere seguiti da un `L`, per esempio `A=3455555L`, come le costanti `float` dovrebbero essere seguite da un `f.`

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole specifiche di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sessione 2.17.

2.11 Operazioni fra variabili Booleane

Le variabili *Booleane* assumono solo due valori: `TRUE` o `FALSE` (vero o falso). Nonostante questa limitazione vengono tipicamente rappresentate in C/C++ con numeri interi. Le variabili intere sono considerate vere se diverse da `0`, e false quando sono uguali a 0. Fra questo tipo di variabili si possono utilizzare gli operatori dell'algebra di Bool: and, or, not, xor (or esclusivo).
In C/C++ questi operatori logici devono essere specificati con i simboli `&&`, `||`, e `!` (and, or e not, rispettivamente). Per esempio:

```
int a=1, b=0, c;
c = a && b;  // c = a and b
```

Il valore assegnato alla variabile `c` e' pari a `FALSE`, cioè 0, in accordo alla semantica dell'operatore di congiunzione discussa nella parte relativa ai sistemi di numerazione.

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole specifiche di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sessione 2.17.

2.12 Operazioni a livello dei bit

In C/C++ sono possibili anche operazioni binarie sui singoli bit di variabili interi con e senza segno. Tali operazioni sono: and sui bit, or sui bit, complemento a 1, xor sui bit, con i seguenti operatori: `&, |, ~, ^`.

```
int a = 152, b = 23., c;
c = a | b;
```

Il valore assegnato alla variabile `c` e' pari a 143, in accordo alla semantica dell'operatore or fra numeri binari dove 152 in binario assume la forma 10011000, mentre 23 in binario e' 10111.

Su numeri binari sono possibili anche le operazioni di scorrimento a destra e sinistra: <<, >>, dette di "shift". Per esempio:

```
int a = 17, b = 13, c, d;
c = a >> 2; // scorrimento a sinistra di due bit
d = b <<1; // scorrimento a destra di un bit
```

Il valore assegnato a `c` e' pari a 4, cioè `00000100`, poiché, 17 in binario e' `00010001`. A `b` viene assegnato 26, cioè `00011010`, poiché, 13 in binario e' `00001101`.

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole specifiche di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sessione 2.17.

2.13 Numeri Reali

I numeri reali possono essere utilizzati per mezzo della rappresentazione in virgola mobile.

I numeri reali possono essere scritti con la notazione tradizionale, parte intera e parte frazionaria o in notazione scientifica:

```
float a, b, c;
a = -134.566788;
b = -45.4E+12;
b = -44.4 e -12;
c = 34.567777f
```

La seconda notazione `-45.4E+12` significa -45.4 * 10^{12}. Si possono scrivere numeri in notazione scientifica con esponente negativo in accordo alla precisione della rappresentazione. La 'e' che rappresenta l'esponente può essere sia minuscola sia maiuscola e vi possono essere spazi fra il numero e

l'esponente. Si può utilizzare anche il carattere `f` come ultimo carattere del numero per specificare che si tratta di un `float`.

2.14 Operazioni fra numeri reali

Per tali tipi di dato sono definite le operazioni di somma, sottrazione, moltiplicazione e divisione, con gli operatori: `+, -, *` e `/` come per gli interi. L'unica differenza e' la semantica delle operazioni in termini di precisione e approssimazioni come e' stato mostrato nella parte relativa ai sistemi di numerazione. La divisione fra numeri in virgola mobile non produce il troncamento come fra gli interi ma può ovviamente produrre degli errori dovuti ai limiti della rappresentazione (si veda la parte relativa alla rappresentazione di numeri).

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole specifiche di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sezione 2.17.

2.15 Operatori di confronto

Gli operatori di confronto fra numeri si possono applicare a numeri interi, naturali e reali.

Tali operatori in C/C++ sono: `<` (minore), `<=` (minore uguale), `>` (maggiore), `>=` (maggiore uguale), `!=` (diverso), `==` (uguale). Per esempio:

```
int a = 12, d=25, f;
float b=1.2, c=0.0;

f = a > b && c <= d ;
```

Il valore assegnato ad `f` e' pari a 1, cioè `TRUE`.
A sinistra del segno di assegnazione si ha un'espressione *Booleana*.

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sezione 2.17.

2.16 Le espressioni algebriche e precedenze

Le espressioni algebriche possono contenere variabili e costanti numeriche di vario tipo. In questo contesto, in C/C++, caratteri e variabili *Booleane* vengono considerate variabili numeriche. Le regole sintattiche delle espressioni si basano sulle regole sintattiche degli operatori. Questi possono essere binari (cioè far riferimento a due operandi) o unari (far riferimento a un solo operando). Le espressioni non possono contenere due operatori consecutivi a meno che non siano separati da parentesi. L'espressione più semplice si può ridurre ad una semplice costante o variabile.

Per la stima di espressioni valgono le regole dell'algebra. Prima vengono eseguite le operazioni di prodotto e divisione e quindi quelle di somma e sottrazione. Nei casi in cui l'ordine può portare a produrre

risultati diversi e'possibile utilizzare le parentesi tonde, ( ). Le operazioni racchiuse da parentesi vengono eseguite prima delle altre, partendo dalle parentesi piu' interne all'espressione. Per esempio:

```
int a = 15, b = 7, c;
float d = 13.3, e = 12.5, f;

c = a*2/b+3;         c = a*2/(b+3);
f = a*d+3/b;         f = a*(d+3)/b;         f = (a*d+3);
```

La prima e la seconda assegnazione della variabile `c` producono risultati diversi: 7 e 3. Nel secondo caso si arriva a 3 eseguendo 30/10. Le assegnazioni di `f` producono risultati diversi. Lo studente calcoli per esercizio i valori assegnati alla variabile nei tre casi.

Nelle espressioni i vari operatori vengono eseguiti in base a delle regole di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sessione 2.17.

2.17 Precedenze fra operatori

Nella seguente Tabella sono riportate le relazioni di precedenza del linguaggio C/C++. Nella Terza colonna, oltre all'azione e' riportata anche la modalità con la quale vengono associati gli operandi agli operatori. Questa può essere da sinistra a destra o da destra a sinistra. Per esempio, se un operatore associa da sinistra a destra significa che prima viene stimata l'espressione che produce l'operando di sinistra e quindi si passa a stimare l'espressione che produce l'operando di destra. Quando queste due parti sono ridotte a valori allora si procede ad effettuare l'operazione. Se vi e' più di un'operazione dello stesso tipo si procede nell'ordine stabilito.

Livello di precedenza	Operatori	Descrizione
0	`::`	visibilità (class name:: member)
1	`() [] -> .`	Precedenza, subscritto di matrice/array, membro di struttura da puntatore (metodo da puntatore), membro di struttura da struttura (metodo da oggetto). (da sinistra a destra)
2	`! ~ ++ -- - + (type) * & sizeof() delete new`	Not logico, complemento a uno, incremento, decremento, cambio di segno, segno positivo, casting, indirezione (variabile puntata da), indirizzo di, dimensione di, delete, new. (da destra a sinistra)
3	`* / %`	Prodotto, divisione, modulo. (da sinistra a destra)
4	`+ -`	Somma (più), differenza (meno). (da sinistra a destra)
5	`<< >>`	Shift a sinistra sui bit, shift a destra sui bit. (da sinistra a destra) (utilizzati anche per lo stream di ingresso e uscita)
6	`< <= > >=`	Minore?, minore uguale?, maggiore?, maggiore uguale? (da sinistra a destra)
7	`== !=`	Uguale?, diverso? (da sinistra a destra)
8	`&`	And sui bit. (da sinistra a destra)
9	`^`	Xor (or esclusivo) sui bit. (da sinistra a destra)
10	`|`	Or sui bit. (da sinistra a destra)
11	`&&`	And logico. (da sinistra a destra)
12	`||`	Or logico. (da sinistra a destra)
13	`?:`	espressione condizionale. (da destra a sinistra)
14	`= += -= *= /= %= &= ^= |= <<= >>=`	Assegnazioni con operazione. (da destra a sinistra)
15	`,`	Virgola come separatore di istruzioni. (da sinistra a destra)

2.18 Caratteri e operazioni

La gestione di caratteri alfanumerici in programmi e' senza dubbio una delle caratteristiche più interessanti. Ogni carattere può assumere uno dei simboli possibili della tabella ASCII semplicemente assumendone la codifica binaria a 8 bit.
Per l'assegnazione di un carattere ad una variabile di tipo `char` si deve utilizzare una specifica notazione:

```
char ca;
ca = 'a';
```

In questo caso e' stato assegnato il carattere `'a'` alla variabile `ca`. Questa conterrà la codifica ASCII di tale carattere. La stessa assegnazione poteva essere effettuata indicando il codice ASCII decimale del carattere `'a'` che risulta essere `97`, oppure in esadecimale (`0x61`) o in ottale.
Pertanto le seguenti assegnazioni sono del tutto equivalenti:

```
ca = '\134';     ca = '\0206';    ca = '\0x86';
```

Il carattere vuoto, non uno spazio, ma proprio il carattere che non esiste come carattere ha come codifica lo 0.

In C, tutti i tipi, ad esclusione di quelli che modellano i numeri razionali, sono considerati interi, anche i caratteri. Pertanto sono possibili le seguenti equivalenti assegnazioni: `A=0x41` nel caso che `41` sia un numero esadecimale, `A=65` nel caso che 65 sia un numero decimale, `A='A'` in forma di carattere, `A='\0101'` da ottale in forma di carattere, `A='\0x41'` da esadecimale in forma di carattere.

Le operazioni consentite fra caratteri sono praticamente tutte quelle consentite per gli interi anche se molte di queste non hanno senso per i caratteri.

2.19 La dichiarazione di costanti

In C/C++ la dichiarazione di costanti può essere effettuata in due modi diversi. Il primo permette di definire delle costanti tipizzate ed e' il modo più corretto per definire costanti che possono essere utilizzate all'interno di espressioni.

```
const int a = 56444;
const float pigreco = 3.14;
const char lettera_a = 'a';
const unsigned long = -456000000;
```

Le costanti possono essere utilizzate in espressioni e assegnazioni ma non possono figurare a sinistra del segno di assegnazione. In tal caso, il compilatore manifesta un errore sintattico grave (severe).

Le costanti si possono definire anche tramite il costrutto `define` del preprocessore. Con il seguente esempio viene definita la costate VERO e gli viene assegnato il valore 1. Costanti definite con questa modalità non hanno un tipo definito in modo formale e pertanto il tipo si evince dal valore che gli viene assegnato. Il compilatore in questo caso non e' in grado di effettuare un vero e proprio controllo di consistenza del tipo.

```
#define VERO 1
```

In base a questa istruzione il preprocessore sostituisce il valore 1 al simbolo VERO nel programma prima della compilazione vera e propria. Tali tipi di costanti sono anche dette costanti simboliche e possono essere utilizzate per definire dei simboli complessi con parametri detti macro. Si veda Sessione 10.18.

2.20 Istruzioni di incremento e decremento

In molti linguaggi vi sono delle specifiche istruzioni per effettuare incrementi o decrementi di variabili intere o naturali. In C tali operazioni possono essere effettuate tramite degli operatori unari.

Operatore di incremento ++: quest'operatore viene utilizzato per incrementare di un'unita' una variabile. Per esempio con `i++;` si incrementi la variabile i di un'unita'. Questo significa che `i++` e' del tutto equivalente a `i=i+1;` quest'affermazione e' vera ma l'operatore ++ può essere utilizzato anche

all'interno di espressioni e cambia il proprio significato se posto prima o dopo la variabile da incrementare. Per esempio se `n==5` allora con l'istruzione:

```
x = n++;
```

si assegna ad `x` il valore 5 e quindi `n` viene incrementata a 6, dopo il calcolo dell'espressione. Se `n==5` allora con l'istruzione:

```
x = ++n;
```

si assegna ad `x` il valore 6: `n` viene prima incrementato puoi viene utilizzato nell'espressione.

Operatore di decremento --: questo operatore viene utilizzato per decrementare di un'unita' una variabile intera. Per esempio con `i--;` si decrementa la variabile `i` di un'unita'. Questo significa che `i--` e' del tutto equivalente a `i=i-1`; Quest'affermazione e' vera ma l'operatore `--` può essere utilizzato anche all'interno di espressioni e cambia il proprio significato se posto prima o dopo la variabile da incrementare come per l'operatore ++.

2.21 Assegnazioni con operatore

In C/C++ vi sono degli operatori composti da un'assegnazione e da un operatore algebrico o Booleano. Tali operatori sono delle forme contratte che se utilizzate nella scrittura dei programmi permettono al compilatore di produrre il codice eseguibile più efficiente. Questo significa che non e' detto che il loro utilizzo dia sempre luogo a programmi più veloci. Di seguito sono riportate le forme contratte e il loro significato riportato a destra in un commento:

```
a += b;    // a = a + b; somma con accumulo
a -= b;    // a = a – b; sottrazione cumulativa
a /= b;    // a = a / b; divisione
a *= b;   // a = a * b; prodottoria
a %= b;   // a = a % b; modulo
a &= b;   // a = a & b; and sui bit
a ^= b;   // a = a ^ b; xor sui bit
a |= b;   // a = a | b; or sui bit
a <<= b;  // a = a << b;  shift a sinistra di b bit
a >>= b;  // a = a >> b;  shift a destra di b bit
```

2.22 Tipi enumerativi

Oltre ai tipi numerici e caratteri si possono definire dei tipi scalari enumerativi. Questi sono definiti per mezzo della specifica dell'elenco dei possibili valori che possono avere. Il nome dell'enumerazione e' praticamente un tipo per il quale sono definite solo delle semplici operazioni di assegnazione e confronto.

```
enum Bool { FALSE, TRUE};
enum Giorni { LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI,
              SABATO DOMENICA};
```

```
enum Mesi { GEN=1, FEB, MAR, APR, MAG, GIU, LUG, AGO, SET,
            OTT, NOV, DIC};
enum valori_di_stato { BGT = 123, BHG = 345, JKK = 45 };
enum GN { GIO='G', NOT='N' };
```

Dai vari esempi e' chiaro come si possano definire tipi enumerativi che si basano su tipi di dati diversi: interi e caratteri. Nel tipo enumerativo Bool viene assegnato 0 a FALSE e 1 a TRUE. Nel secondo caso ai giorni vengono assegnati i numeri da 0 a 6. Per il tipo enumerativo Mesi, sono assegnati i valori da 1 a 12 ai nomi dei vari mesi.

I valori del tipo che codifica i `valori_di_stato` e' un esempio interessante di un tipo enumerativo nel quale i singoli valori ammissibili non sono ordinati in base alla loro posizione nella definizione. Infine, l'ultimo assegna un carattere ad ogni elemento del tipo enumerativo.

L'utilizzo dei tipi enumerativi si effettua come se questi fossero tipi di variabili:

```
Mesi mes = GEN;
int a;
mes = FEB;
a = MAR;
mes = 3;
```

Nell'esempio l'ultima assegnazione e' un errore sintattico poiché la conversione da intero a enumerate non e' automatica, mentre la conversione da un tipo enumerativo in intero e' automatica. La conversione automatica viene detta anche implicita.

Non e' possibile cambiare la definizione di un tipo enumerativo durante l'esecuzione del programma.

2.23 Controllo dei tipi

Uno dei compiti del compilatore e' quello di controllare la congruenza dei tipi che sono stati utilizzati in base alle operazioni che sono disponili e lecite per loro. Non deve essere possibile per un compilatore accettare un programma dove si utilizzano degli operatori su variabili per le quali non sono stati definiti. Il controllo di congruenza del tipo deve essere effettuato tutte le volte che un dato (variabile o costante che sia) e' utilizzato in espressioni e/o assegnazioni. Le regole che governano questo tipo di controllo sono regole sintattiche.

Il controllo di congruenza dei tipi deve essere effettuato anche quando si utilizzano delle funzioni di libreria. Queste accettano parametri di tipi specificati. Il loro utilizzo con variabili di tipi non congruenti può dare luogo a problemi di compilazione e/o di esecuzione. Queste problematiche saranno chiarite in seguito.

Il controllo dei tipi viene *detto type checking* e può essere effettuato in fase di compilazione (*static type checking*) o durante l'esecuzione del programma (*dynamic type checking*) quando l'operazione viene eseguita.

Questo tipo di controllo può essere necessario per quei linguaggi in cui non si ha una dichiarazione preventiva del tipo della variabile e per verificare la congruenza di operazioni su variabili e/o puntatori che possono assumere tipi diversi durante l'esecuzione del programma.
Questo ultimo controllo di tipo sarà meglio trattato in seguito.

Per il momento ci soffermiamo solo al controllo statico dei tipi che viene effettuato durante la compilazione e può evidenziare errori sintattici di vario genere. In certi casi sono possibile delle conversioni come descritto nella seguente sessione.

2.24 Conversioni fra tipi, casting implicito ed esplicito

Le variabili e le costanti appartenenti a tipi diversi possono in vari casi essere convertite da un tipo all'altro senza perdita d'informazione. Questa possibilità dipende dai tipi di dati coinvolti, dal valore del dato, dalla rappresentazione che viene utilizzata nello specifico linguaggio e/o compilatore.

Le conversioni fra tipi si possono classificare in implicite ed esplicite. Le conversioni si dicono implicite quando il compilatore non richiede nessun intervento da parte del programmatore per specificare il tipo di conversione. Mentre si dicono esplicite quando e' necessario specificare in dettaglio la conversione che deve essere effettuata. Dai realizzatori dei compilatori vengono utilizzate le conversioni implicite per quei casi in cui non si ha perdita di informazione. Per esempio nel caso di conversione da numero carattere a intero, o da enumerativo a intero.
Spesso i compilatori evidenziano la presenza di conversioni implicite con dei messaggi di attenzione, *Warning*. Questo deve essere considerato come un errore non grave ma che comunque necessita l'attenzione del programmatore per definire in dettaglio che tipo di conversione deve essere effettuata e se e' lecito che ne venga effettuata una oppure il *Warning* e' stato provocato da una leggerezza del programmatore che ha utilizzato una variabile di tipo non appropriato.

Le conversioni esplicite si effettuano principalmente in due modi; per mezzo:

- Dell'operatore di cast composto dal tipo della variabile verso la quale si vuole effettuare la conversione, racchiuso fra parentesi tonde: (`char`), conversione verso un carattere.
- Di specifiche funzioni di libreria. Per questa modalità si veda il manuale di riferimento del vostro compilatore nel quale troverete la lista delle funzioni di libreria disponibili che possono essere utilizzate per le conversioni. Per una tabella ridotta di veda Sessione 3.5.

Di seguito sono riportati alcuni casi di conversioni implicite ed esplicite.

```
int a = 23, b = 45;
float e = 45.3, f = 0.34;

a = e;                    // errore conversione non consentita
a = (int) e ;       // conversione esplicita
a = e*f;            // errore conversione non consentita
a = (int)e * f ;    // conversione esplicita
a = (int) (e * f);  // conversione esplicita
f = a + b;                // conversione implicita a float
f = a * 45.3;       // conversione implicita a float
f = a * (int) 45.3 ;      // conv. esplicita a int ed implicita a float
```

Negli esempi riportati sopra vi sono anche alcuni predicati non corretti dal punto di vista sintattico per la mancanza di una conversione esplicita. La quarta e quinta assegnazione della variabile `a` danno luogo a diversi valori. Questo e' dovuto alla priorità dell'operatore di cast rispetto a quello di prodotto come riportato in sessione 2.17 ove sono esposte in una tabelle le precedenze fra operatori.

3 Modulo: Programmi

Lo scopo di questo modulo e' dare un'idea generale sulla realizzazione di un programma. In questo modo lo studente e' in grado di procedere a sperimentare la manipolazione delle variabili e la loro dichiarazione con una strumento (il compilatore) che puo' aiutarlo a verificare la correttezza di cio' che viene prodotto.

Per questo motivo viene presentato un primo programma in C ed alcuni rudimenti per la stampa a schermo di messaggi e dati provenienti dal programma. Lo studente potra' scrivere programmi composti da sequenze di istruzioni di assegnazione con relative espressioni e stampare i risultati a schermo.

Viene inoltre mostrato come si possono includere dei commenti nei programmi e come si puo' fare uso delle potenti librerie del linguaggio C/C++. Viene infine suggerita una struttura di massima per la realizzazione di programmi.

3.1 Il primo programma

La realizzazione di un piccolo programma in C/C++ e' molto semplice:

```
void main(void)
{
int a;
float b;
a=15;
b = a * 3.3;
printf("il valore di a = %d, il valore di b : %f", a, b);
}
```

Il costrutto `void main(void)` marca l'inizio del programma. Dopo il marcatore d'inizio programma il programma vero e proprio inizia con la prima parentesi graffa aperta, {, e termina con un corrispondente parentesi graffa chiusa, }. In C/C++ le parentesi graffe identificano il costrutto sequenza come sara' mostrato in dettaglio in seguito. Per il momento la coppia di parentesi graffe, { }, delimitano semplicemente il corpo del programma. All'interno del programma le varie istruzioni che lo compongono. In questo caso ci sono due dichiarazioni, due assegnazioni ed un'istruzione di uscita.

Il programma può essere compilato ed eseguito provocando la seguente stampa a schermo:

```
il valore di a = 15, il valore di b : 49.5000
```

3.2 Stampa a schermo, rudimenti

In questo programma e' stata utilizzata la funzione di libreria `printf()`. Tale funzione può essere considerata come un'istruzione di uscita. Questa permette la stampa a schermo di variabili tramite una conversione di formato specificata fra virgolette. Tale formato può includere sia caratteri ASCII, per produrre parole sullo schermo, che marcatori per convertire in formato ASCII i numeri. Nell'esempio soprastante viene utilizzato `"%d"` per convertire un intero nella sua rappresentazione ASCII e `"%f"` per i `float`. Tale conversione permette di visualizzare a schermo tramite caratteri ASCII il valore di una memoria rappresentata in complemento a 2. Si veda in seguito la descrizione dettagliata dei convertitori di formato per la funzione `printf`(). Questa funzione permette di effettuare delle operazioni di conversione e stampa in base a certi parametri che gli vengono passati fra le sue parentesi tonde. L'utilizzo di funzioni e' uno degli aspetti che caratterizzano maggiormente i linguaggi come il C, Pascal, etc. Questi linguaggi hanno possibilità di integrare e quindi utilizzare collezioni di funzioni prodotti da altri programmatori nei programmi personali. Queste collezioni di funzioni sono tipicamente chiamate librerie.

3.3 Stampa tramite `printf()`, formati

Una delle caratteristiche fondamentali dei calcolatori elettronici e' quella di memorizzare dati, effettuare calcoli e produrre dei risultati. A tale scopo e' di fondamentale importanza l'utilizzo di funzioni per acquisire dati dall'esterno e per produrre i risultati una volta calcolati. Nei moderni calcolatori i dati possono essere acquisiti tramite tastiera o passati tramite file, o altri mezzi. Mentre il metodo più semplice per produrre i risultati e' senza dubbio lo schermo del computer, l'interfaccia grafica, nei più moderni elaboratori.

Per quanto riguarda il linguaggio C, alla stampa di una variabile a schermo per visualizzarne il valore corrisponde sempre una conversione di formato. Pertanto, esiste una notazione per definire la conversione di formato quando viene visualizzato il valore di una variabile per mezzo di funzioni tipo `printf`() e `fprintf`(). A questo fine si può consultare la seguente Tabella. Ogni descrittore di formato di uscita può essere costituito da cinque elementi nella forma `%knqT`, dove tali componenti hanno il seguente significato.

- Il simbolo di percentuale identifica il convertitore di formato nella stringa di stampa.
- `k` può essere presente o meno. In questo campo possono essere posti uno o più flag: `-` specifica l'aggiustamento a sinistra del numero; `+` impone la visualizzazione del segno +. Per altri segni particolari si consulti il manuale dello specifico compilatore in uso.
- `n` può essere presente o meno e rappresenta il formato di presentazione del numero. Se il numero e' intero, `n` può essere un numero che specifica in caratteri lo spazio da destinare alla variabile. Se il numero non e' un intero può essere nella forma `a.b`, dove `a` e' lo spazio in caratteri e `b` il numero di cifre dopo la virgola.
- `q`, se presente, può essere uguale a `h` per `short` o `unsigned short`, `l` per `long` o `unsigned long`, e `L` per `long double`;
- `T` indica il tipo come riportato in Tabella.

T	Argomento	convertito in
d, i	int	Decimale con segno
O	int	Ottale senza segno, senza O
x, X	int	Esadecimale senza segno (senza 0x o 0X), con X si ha ABCDEF anziché abcdef
U	int	Decimale senza segno
C	int	carattere (conversione implicita con unsigned char)
C	unsigned char	carattere
S	char*	stringa di caratteri fino al prossimo \0
F	double	numero reale nella forma *mmm.ddd*
F	float	numero reale nella forma *mmm.ddd*
e, E	double	numero reale nella forma scientifica *m.ddde±xx*, *m.dddE±xx*
e, E	float	numero reale nella forma scientifica *m.ddde±xx, m.dddE±xx*
g, G	double	come %e,%E se l'esponente e' minore di -4 altrimenti come %f
g, G	float	come %e,%E se l'esponente e' minore di -4 altrimenti come %f
P	void*	puntatore far
N	int*	puntatore near (senza segmento)
%		carattere %

Si noti la possibilità di stampare numeri in varie forme ma anche caratteri.

E' infatti molto importante poter stampare a schermo non solo i risultati numerici ma anche alcuni commenti testuali per poter meglio identificare i singoli dati sul dispositivo di uscita. Per questo motivo la funzione `printf()` e' sufficientemente flessibile da poter essere utilizzata per stampare sia la descrizione che i valori numerici con la stessa chiamata.

3.4 I commenti multilinea e di linea

Durante la stesura di programmi e'buona regola inserire dei commenti in linguaggio naturale per descrivere le istruzioni stesse del programma.

Tali commenti possono essere inseriti direttamente nel codice ma devono essere delimitati da opportuni marcatori in modo che il compilatore li possa distinguere dalle istruzioni. I commenti non hanno nessun effetto sull'esecuzione del programma nel senso che non portano a nessuna traduzione in linguaggio macchina, vengono direttamente ignorati dal compilatore o ancor prima in una fase di preprocessazione del codice.

I commenti possono essere monolinea (o semplicemente "di linea") e multilinea.

I commenti monolinea iniziano con un marcatore e finiscono con la fine della linea di testo nel programma. Per esempio il tipico commento monolinea del C++ viene effettuato tramite due barre, //. Di seguito viene riportato un'esempio del suo utilizzo:

```
int alfa; // valore di controllo
……
a = alfa * 12 +3 ; // stima del coefficiente a
```

I commenti multilinea possono iniziare in qualsiasi punto del programma con unmarcatore di inizio commento e terminano con un marcatore di fine commento. In C/C++, i marcatori di inizio e fine commento sono rispettivamente le coppie di caratteri /* ed */.
Questi commenti possono racchiudere anche commenti di linea e quindi hanno una maggiore priorità nei loro riguardi. Per esempio:

```
int alfa; // valore di controllo
……
/* stima di a
a = alfa * 12 +3 ; // stima del coefficiente a
*/
```

In questo modo, l'assegnazione viene commentata.

In generale i commenti di linea e multilinea possono contenere anche istruzioni. Questo e' particolarmente utile quando il programmatore ha necessita' di escludere alcune istruzioni dal programma senza per questo doverle cancellare se non e' del tutto sicuro.

I commenti devono essere utilizzati per inserire delle descrizioni in linguaggio naturale del programma stesso. Un buon programma, un programma che può essere facilmente manutenuto, per correggere eventuali problemi o per soddisfare nuove necessita' deve essere abbondantemente commentato. Una semplice regola e' di inserire almeno un commento per ogni istruzione significativa del programma. Istruzioni sequenziali che ripetono la stessa cosa su variabili diverse non necessitano di commenti distinti.

3.5 Le funzioni di libreria

Per poter utilizzare le funzioni di libreria vi e' spesso la necessita' di includere nel proprio programma un file di definizioni correlate alla libreria che s'intende utilizzare. Tale inclusione deve essere dichiarata in modo esplicito all'interno del programma con un'istruzione che viene interpretata dal preprocessore del compilatore C/C++.
I file contenenti le definizioni della libreria sono detti anche file *Header* (intestazione) e possono avere estensione `h`, `hxx`, oppure `hpp`.

Questi file contengono tipicamente:

- definizioni di tipi,
- prototipi di funzioni (si veda i sottoprogrammi),
- dichiarazioni di variabili generali della libreria,
- costanti simboliche per gestire le funzioni della libreria.

Tali file possono contenere anche sottoprogrammi o comunque istruzioni. Quest'opzione e' da evitare nel modo più assoluto al fine di rendere il vostro programma comprensibile e le vostre procedure riutilizzabili.

3.6 Le funzioni di libreria: loro inclusione

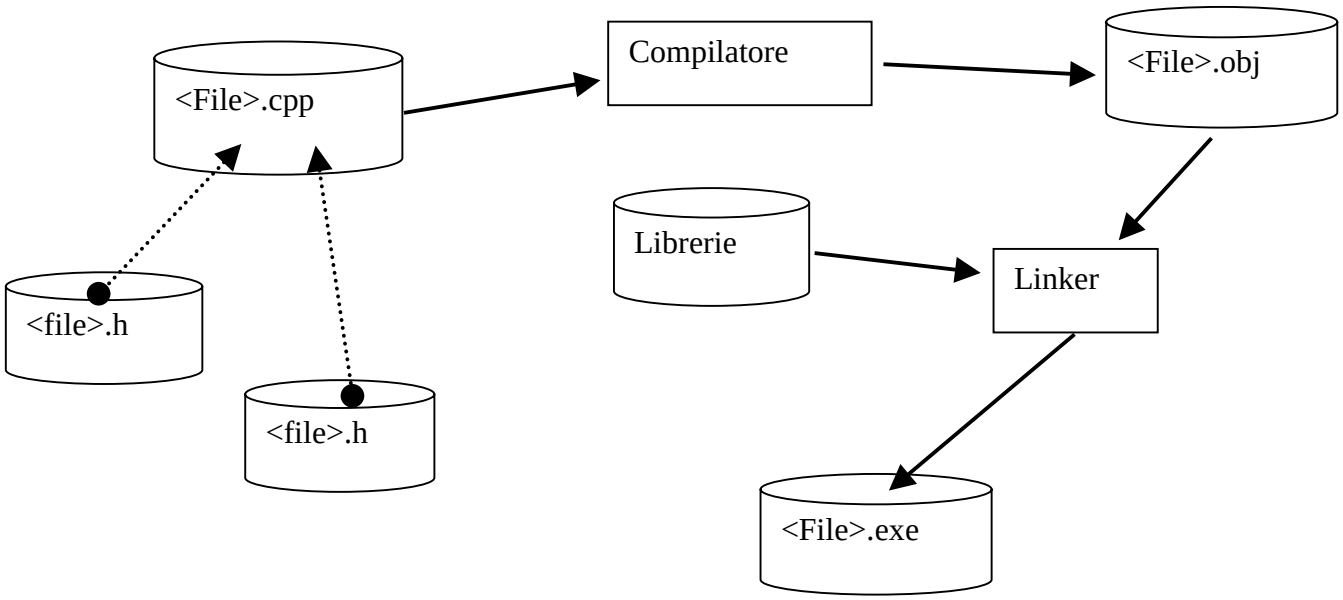
Con le seguenti istruzioni di inclusione si possono includere i file di dichiarazioni (*header file*) delle librerie più importanti:

```
#include "stdlib.h" // libreria standard
#include "stdio.h"  // libreria di ingresso/uscita
#include "string.h" // per la manipolazione di stringhe
#include "math.h"   // per le funzioni matematiche
```

In C/C++, la loro inclusione può essere effettuata in qualsiasi punto del programma. Si consiglia comunque di utilizzare tale istruzione del preprocessore solo nelle prime istruzioni di ogni file del vostro programma per evitare conflitti e confusione su quello che viene effettuato nel file stesso.

L'inclusione dei file dichiarati con `include` viene effettuata durante la fase di proprocessazione. Questa e' una fase preliminare alla compilazione ed e' direttamente effettuata dal compilatore stesso, il programmatore lancia la compilazione, e' il compilatore stesso che prima di effettuare la compilazione vera e propria esegue la preprocessazione. Il preprocessore include i file dichiarati in `include` direttamente nel file che processa. Il file con gli *header* inclusi viene in seguito compilato dal compilatore vero e proprio.

Dopo la fase di compilazione si passa alla fase di collegamento/link durante la quale i vari moduli del programma e le librerie vengono collegati per formate il programma eseguibile. In questa fase vengono collegate le librerie dichiarate al programma realizzato. Tali librerie sono dei file contenenti svariate funzioni in codice oggetto.
Per il programmatore e' possibile creare delle librerie e dei corrispondenti file "*header*" da utilizzare in altri programmi.



3.7 Funzioni di libreria

Nella tabella seguente sono riportate le principali funzioni di libreria tipicamente messe a disposizione dalla maggiore parte dei compilatori C/C++ per effettuare conversioni da tipi, per il calcolo di funzioni matematiche, e per la manipolazione di stringhe. Le funzioni di libreria sono disponibili per essere utilizzate solo dopo aver incluso la loro dichiarazione di prototipo. Si veda la definizione di sottoprogrammi e le istruzioni `#include` del preprocessore.

Funzione	*Descrizione*
`double asin(double)`	calcolo della funzione seno
`double atan(double)`	calcolo dell'arcotangente
`double cos(double)`	calcolo della funzione coseno
`double exp(double)`	calcolo dell'esponenziale
`double log(double)`	calcolo del logaritmo
`double pow(double)`	calcolo del quadrato di un numero
`double sqrt(double)`	calcolo della radice quadrata
`double tan(double)`	calcolo della tangente
`float atof(char *)`	conversione da stringa a float
`float fabs(float)`	calcolo del valore assoluto
`float rand (float)`	generazione di un numero casuale
`int atoi(char *)`	conversione da stringa a int
`int sizeof(<type>)`	stima in byte delle dimensioni di un tipo
`int strlen(char *)`	stima in byte della lunghezza di una stringa
`long atol(char *)`	conversione da stringa a long
`int rand(void)`	Generazione di numeri casuali
`char * strcpy(char *, char *)`	Copia di stringhe
`char * strcmp(char *, char *)`	Confronto di stringhe
`char * strcat(char *, char *)`	Concatenazione di stringhe
`char * sprintf()`	Conversione da dato numerico a stringa

3.8 La struttura dei programmi

La struttura dei programmi in C/C++ può essere di vario tipo. Generalmente i programmi sono composti da una certo numeri di file distinti contenenti istruzioni, dichiarazioni, definizioni, etc.
La seguente struttura per i programmi ne facilita la lettura ma non e' certamente l'unica a poter essere utilizzata.

```
Include di file header di librerie
Definizioni di classi e tipi
Prototipi di funzioni
dichiarazioni di variabili globali // da evitare
dichiarazioni di variabili globali di altri moduli // extern
```

```
main() // programma pincipale
{
}
```

```
procedure, e metodi.
```

Questa semplice struttura non e' utilizzabile quando si hanno programmi di un certa dimensione, anche se di soli pochi Kbyte. In tal caso si consiglia di dividere il programma sorgente in vari file. Una buona regola e' di avere un file di definizione per ogni nuovo tipo di variabile, come classi, etc. E realizzare un corrispondente file di codice contente la realizzazione dei metodi corrispondenti alla classe.

Le parti marcate in blu saranno discusse in seguito, mentre per le parti in rosso si rimanda ad un corso di programmazione di secondo livello.

3.9 Esercizi suggeriti

Realizzare un programma che permetta di calcolare e stampare a schermo il risultato delle seguenti operazioni, dato a=34.4, b =19, c=10, d=9:

$$a^b \sqrt{c + 23 - d}$$

$$2^c \sqrt{c - d} + e^{(a+b)}$$

Sin(a+b) + b c

4 Modulo: Gli array ed I vettori

In questo modulo sono presentati gli array.

Un *array* e' un insieme omogeneo di componenti. Omogeneo nel senso che tutte le componenti sono dello stesso tipo. In tale tipo di struttura dati e' possibile accedere ad ogni singolo elemento in modo diretto tramite un'operazione di selezione. Tale tipo di organizzazione viene detta ad accesso casuale o *random*.

In questo ambito viene data particolare enfasi alla definizione e all'uso di vettori e al loro impiego per la gestione delle stringhe di caratteri.

4.1 Vettori

Quando un *array* e' monodimensionale viene comunemente chiamato vettore. In C/C++, la dichiarazione di un vettore può essere eseguita semplicemente:

```
#define DIMENSIONE 3

……..

int a[4];
float b[7];
double ef[DIMENSIONE], d;
char dop[DIMENSIONE];
```

Nell'esempio riportato sono definiti vari vettori composti da elementi di diverso tipo e di diverse dimensioni. La dimensione del vettore e' specificata fra parentesi quadre. Nelle ultime due righe dell'esempio e' stata utilizzata una definizione di costante simbolica per impostare la dimensione del vettore. Questo permette di realizzare programmi che dipendono da parametri simbolici che possono essere utilizzati in vari punti del programma stesso.

4.2 Inizializzazione dei vettori

Anche i vettori possono essere inizializzati come le variabili singole. In tal caso si deve utilizzare la seguente sintassi:

```
int a[4] = {1, 2, 3, 5};
float b[7] = { 34.0, 67.0, 45.4, 56E-2, 45.0, 34, 78 };
double ef[DIMENSIONE] = {10.3, 45.5, 553333332222.4 } , d;
```

Le inizializzazioni riportate sopra devono avere un numero congruente di elementi rispetto alla dichiarazione della dimensione della variabile vettoriale stessa. Si possono definire dei vettori con una data di inizializzazione in modo che acquistino direttamente la dimensione in base al numero di componenti forniti durante la loro inizializzazione.

```
int a[] = {1, 2, 3, 5};
float b[] = { 34.0, 67.0, 45.4, 56E-2, 45.0, 34, 78 };
double ef[]= {10.3, 45.5, 553333332222.4 } , d;
```

Questo si effettua semplicemente non definendo la dimensione del vettore, lasciando vuota la dimensione del vettore, `[ ]`.
Tale modalità di dichiarazione non può essere utilizzata quando il vettore non ha i valori di inizializzazione.

4.3 Accesso ai singoli elementi degli array

Per accedere ai singoli elementi del vettore si deve utilizzare un indice che identifica l'elemento del vettore considerato. Per esempio con `a[3]`, si identifica l'elemento di indice 3 del vettore. Le singole componenti dei vettori possono essere utilizzate come componenti scalari in espressioni e a sinistra di operazioni di assegnazione.

```
int k;
……….
a[3] = k * 23 + 7;
b[2] = a[0]*b[1]+23.5;
printf("il secondo elemento di a = %d", a[1]);
```

In C/C++, gli indici dei vettori partono dal valore 0 alla loro dimensione diminuita di una unita'. Questo significa che in un vettore di N elementi il primo elemento e' l'elemento di indice 0 mentre l'ultimo e' quello di indice N-1. L'ultima istruzione dell'esempio provoca la stampa di una scritta in caratteri ASCII seguita del valore dell'elemento di indice 1 del vettore `a`, il secondo elemento del vettore.

```
Il secondo elemento di a = 2
```

4.4 La stringa come vettore di caratteri: strlen()

Nei linguaggi di programmazione e' indispensabile la gestione delle stringhe. Una parola, una frase, un insieme monodimensionale di caratteri ASCII e' una stringa. Le stringhe sono messe a disposizione in varie forme nei vari linguaggi. In alcuni di questi, come il Pascal, il tipo stringa e' un tipo elementare. In C/C++ non esiste il tipo elementare di stringa, per questa ragione le stringhe vengono realizzate come *array* monodimensionali di caratteri.

```
char buf[20] = "Saluti a tutti";
printf("la stringa %s\n",buf);
printf("il terzo carattere %c\n", buf[2]);
```

In questo caso la stringa (vettore di caratteri) `buf` e' composta da 20 caratteri (da 0 a 19). Il primo carattere (di indice 0) e' stato inizializzato con il carattere `'S'`. Si noti come l'inizializzazione di stringhe deve essere effettuata tramite una serie di caratteri delimitati da doppie virgolette. Mentre l'inizializzazione dei singoli caratteri deve essere effettuata con le virgolette singole. L'esempio produce la seguente stampa a video:

```
la stringa Saluti a tutti
il terzo carattere l
```

Nell'esempio, le definizioni di formato della funzione `printf()` presentano `\n` come ultimo carattere che definisce il formato. Questo carattere permette di produrre a stampa un carattere che provoca il ritorno alla linea successiva (*new line*). I caratteri stampati dopo questo carattere vengono posti alla linea successiva rispetto a quelli prima di tale carattere.

4.5 Inizializzazione di stringhe

Nell'esempio iniziale la stringa `buf` e' stata inizializzata con un insieme di caratteri minore rispetto alla sua dimensione, impostata a 20, da 0 a 19. La stringa utilizzata per l'inizializzazione `"Saluti a tutti"` e' composta da 14 caratteri (inclusi gli spazi). Utilizzando la funzione intrinseca `strlen()` e' possibile far calcolare al programma la dimensione della stringa in termini di caratteri.

Anche per quanto riguarda le stringhe e' possibile definire con l'inizializzazione la dimensione del vettore di caratteri, cioè della stringa:

```
char buf[ ] = "Saluti a tutti";
printf("la stringa \"%s\" di %d caratteri\n",buf, strlen(buf));
```

L'esempio produce la seguente stringa a video:

```
la stringa "Saluti a tutti" di 14 caratteri
```

Si noti l'utilizzo dei `\"` per produrre a stampa il carattere `"`. Questo carattere come lo `\n` per l'inserimento di un ritorno alla linea successiva fanno parte di un insieme di caratteri speciali che devono essere utilizzati per la generazione e l'impostazione di stringhe. Si veda sessione 4.7.

4.6 Terminazione di stringhe, carattere di fine stringa

In C/C++, poiché un vettore di caratteri può contenere una stringa di dimensioni minori e' stato necessario indicare le dimensioni della stringa direttamente nella sua rappresentazione interna. Questo approccio e' presente anche in Pascal dove il carattere di indice 0 in una stringa contiene il numero di caratteri della stringa. In C/C++, l'inizializzazione di una stringa aggiunge dopo i caratteri della stringa assegnata un carattere specifico di fine stringa, il carattere di codice ASCII decimale 0. Spesso specificato con '\0'. Pertanto le seguenti due inizializzazioni sono del tutto equivalenti:

```
char buf[ ] = "Saluti";
char buf[ ] = { 'S', 'a', 'l', 'u', 't', 'i', '\0'} ;
```

In entrambe i casi la `buf` e' stata inizializzata con una stringa di 6 caratteri e pertanto il vettore di caratteri viene prodotto di 7 caratteri, 6 di dati e uno di fine stringa. Anche nel caso precedente si ha la stessa operazione:

```
char buf[20] = "Saluti a tutti";
printf("Stringa \"%s\"\n", buf);
printf("carattere 13 %d, %c, %x, %o \n",
                    buf[13], buf[13], buf[13], buf[13]);
printf("carattere 14 %d, %c, %x, %o \n",
                    buf[14], buf[14], buf[14], buf[14]);
```

che provoca la seguente stampa a schermo:

```
Stringa "Saluti a tutti"
carattere 13 105, i, 69, 151
carattere 14 0,  , 0, 0
```

Si noti che sia i caratteri di indice 13 e 14 (contenente il carattere di fine stringa, cioè il carattere nullo) sono stati stampati a video in 4 formati diversi, come codifica ASCII decimale, come carattere, come valore esadecimale della codifica ASCII, come valore ottale della codifica ASCII.

Il terminatore di stringa può essere assegnato ad ogni carattere provocando la terminazione della stringa e quindi la sua interruzione.
Il vettore di caratteri che ospita la stringa conserva la propria dimensione iniziale, e' la stringa in esso contenuta che ha delle dimensioni ristrette rispetto al vettore.

```
char buf[20] = "Saluti a tutti";
buf [5] = '\0';
printf("Stringa \"%s\"\n", buf);
```

Che produce a stampa: `Stringa "Salut".` Si noti che la variable `buf` (come vettore di caratteri) continua ad avere un spazio pontenziale di 20 caratteri.

4.7 Caratteri Speciali in Stringhe

Negli esempi precedenti sono stati utilizzati dei caratteri speciali per la generazione di stringhe. Tali caratteri possono venire inclusi in stringhe per mezzo di simboli composti; come riportato nella seguente tabella.

Simbolo	Descrizione
\n	Nuova linea con ritorno carrello (a capo)
\t	Tabulazione orizzontale
\v	Tabulazione verticale
\b	Spazio indietro
\r	Ritorno carrello
\f	Cambio linea senza ritorno carrello (a capo)
\a	Beep (suono prodotto dal computer)
\\	La barra \
\?	Il carattere ?
\'	Il carattere '

\"	Il carattere "
\0II	Il carattere con codifica ASCII ottale II
\xHH	Il carattere con codifica ASCII esadecimale HH
\0	Il carattere NULL

4.8 Gli array multidimensionali

In C/C++, si possono definire degli *array* multidimensionali. La loro definizione e' molto semplice. Con il seguente esempio viene definita un *array* tridimensionale di interi di dimensioni MxNxO:

```
int mat[M][N][O];
```

Dove M, N, O sono delle variabili simboliche specificate per mezzo di istruzioni del preprocessore #define.
Dato un *array* tridimensionale mat si può accedere alle sue componenti semplicemente conoscendone le loro coordinate. Per esempio la componente mat[2][3][7] identifica l'elemento di coordinate (3,4,8) poiché gli indici negli *array* partono da 0.
In C/C++ si possono definire *array* di tipi di base come di tipi definiti nel programma stesso.

Quando gli *array* sono bidimensionali vengono comunemente chiamati matrici.
In generale gli *array* si possono inizializzare tenendo conto della modalità di organizzazione in memoria di tali componenti. Per esempio, nel caso di una matrice bidimensionale di float, Mat[2][4] con 2 righe ed 4 colonne si ha:

```
float Mat[2][4] = { { 1.0,  2.0,  3.0,  5.0},
                    {11.0, 12.0, 13.0, 15.0} };
```

L'elemento della prima riga e della seconda colonna viene identificato da Mat[0][1] poiché anche nel caso di *array* bidimensionali l'indice parte da 0. Si noti come l'identificazione e la definizione degli *array* concorda con una notazione riga, colonna e quindi e' discorde con la notazione x,y che e' conforme alle notazione utilizzata per l'identificazione delle componenti per mezzo delle coordinate.

5 Modulo: I Costrutti di controllo di base

Nelle precedenti sezioni sono state mostrate le istruzioni di assegnazione e le dichiarazioni di variabili e costanti, con e senza inizializzazioni. Con tali tipi di istruzioni si possono codificare algoritmi semplici che non sono in grado di adattare il loro comportamento in base ai dati del problema.

Per descrivere algoritmi piu' complessi e' necessario poter disporre di istruzioni di controllo.
Queste permettono di controllare il flusso del programma e in base ai valori stessi di variabili definire quali istruzioni devono essere eseguite.

Le istruzioni di controllo possono essere classificate in tre categorie: la `sequenza` e la `selezione` e l'`iterazione`.

In questo modulo sono mostrati i costrutti sequenza e selezione (semplice e composta), mentre i costrutti di iterazione sono mostrati in successivi moduli.

5.1 Il costrutto sequenza

La sequenza non e' una vera istruzione di controllo e' solo una formalizzazione affinché sequenze di istruzioni eseguite in successione possano essere considerate come una singola istruzione dal punto di vista delle operazioni da compiere.

Con il costrutto sequenza viene specificato l'ordine con il quale devono essere eseguite altre istruzioni, mentre i costrutti di selezione possono essere utilizzati per alterare tale ordine. Per esempio, alcune istruzioni possono essere eseguite se si verificano determinate condizioni, mentre altre possono essere poste in alternativa fra di loro. I costrutti di iterazione permettono di definire cicli iterativi di istruzioni.

La sequenza di istruzioni e' il metodo più semplice per definire un flusso di esecuzione di istruzioni di un programma, mediante tale costrutto si definisce semplicemente l'ordine di esecuzione delle istruzioni. Per descrivere lo sviluppo sequenziale di un certo numero di istruzioni di programma tramite il linguaggio C/C++ si utilizza la seguente sintassi in BNF:

```
{
  <Dichiarazioni di Variabili Locali>

  <Prima Istruzione>;
  <Seconda Istruzione>;
  ........
  <Ennesima Istruzione>;
}
```

dove le istruzioni possono essere complesse a piacere. In altri linguaggi, vengono utilizzati delimitatori diversi dalle parentesi graffe per marcare l'inizio e la fine di una sequenza di istruzioni. Per esempio in Pascal si utilizza `begin` e `end`. Una sequenza di istruzioni deve essere considerata come una singola

istruzione (un blocco). Tramite la sequenza si realizza una semplice forma di scomposizione/composizione.

All'interno del gruppo di istruzioni e' possibile definire delle variabili locali, le quali non possono essere utilizzate da istruzioni poste al di fuori delle parentesi. Nel linguaggio C, in genere, le singole istruzioni non necessitano di alcun carattere separatore. Si noti che il carattere `";"` in C viene impiegato come terminatore delle istruzioni.

5.2 I costrutti di selezione

I costrutti di selezione sono particolari istruzioni che permettono di specificare fra più istruzioni o gruppi di istruzioni quelle da eseguire, sulla base del valore assunto di un'espressione o variabile. I costrutti di selezione si possono dividere in:

- selezione semplice con o senza alternativa e
- selezione composta con o senza alternativa.

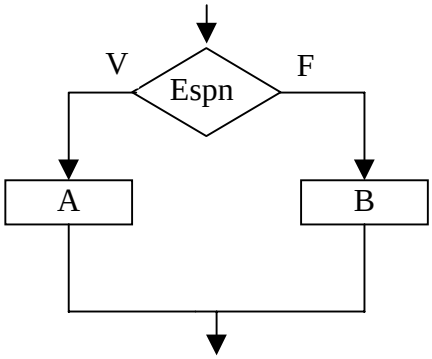
I costrutti di selezione possono essere concatenati per formare costrutti più complessi e potenti.

5.3 Il costrutto di Selezione semplice

In C/C++, la selezione semplice con o senza alternativa si esprime sintatticamente nel modo seguente:

```
if (<Espressione>) <Istruzione A>;
[else <Istruzione B>;]
```

Questa deve essere interpretata come: se (`if`) la condizione Booleana valutata `<Espressione>` risulta vera, allora sara' eseguita L'`<Istruzione A>` altrimenti viene eseguita l'`<Istruzione B>.`



Questa operazione e' rappresentata con il soprastante diagramma di flusso, ove il rombo rappresenta il costrutto di selezione mentre i rettangoli rappresentano le istruzioni.
Se `<Espressione>` assume un valore diverso da zero, la condizione risulta vera per cui viene eseguita l'`<Istruzione A>,` altrimenti viene eseguita l'`<Istruzione B>.` Il linguaggio C/C++, non distingue fra espressioni numeriche e Booleane; un'espressione usata al posto di un predicato logico viene considerata falsa se vale zero, vera in tutti gli altri casi. Pertanto se viene utilizzata come espressione una

assegnazione -- per esempio, `a=45` -- questa sarà sempre vera. Questo e' un errore tipico di chi inizia a programmare in C.

Le istruzioni che vengono messe in esecuzione possono essere assegnazioni, costrutti sequenza contenenti liste di istruzioni, altre istruzioni di selezione ed altri tipi di istruzioni che saranno mostrate in seguito.

Nella sintassi riportata il gruppo `else <Istruzione B>` incluso fra parentesi quadre è opzionale. Cioè l'istruzione `if-else` può essere priva dell'alternativa. Come specificato dalla precedente notazione BNF del costrutto `if` la parte relativa all'istruzione eseguita in alternativa può essere omessa:

```
<Istruzione X>
if (<Espressione>) <Istruzione A>;
<Istruzione Y>
```

In questo caso se l'`<Espressione>` e' soddisfatta vengono eseguite le istruzioni `X, A` ed `Y` in sequenza, altrimenti sono eseguite solo le istruzioni `X` ed `Y`.

In C, le parentesi tonde sono obbligatorie e delimitano l'espressione. Anche in questo caso le istruzioni `A, B, X, Y,` possono essere sia semplici che composte. In mancanza del costrutto di sequenza in associazione all'istruzione di selezione viene eseguita una sola istruzione.

Per esempio:

```
if (a>45)
b = 56 * 78;
c = 34 + 23;
```

In questo caso se `a>45` vengono eseguite entrambe le assegnazioni, mentre se a non e' maggiore di `45` viene eseguita solo la seconda. Nel caso in cui tutte e due le assegnazioni debbano essere eseguite solo se la condizione e' soddisfatta. Il codice doveva essere scritto come segue:

```
if (a>45)
  {
  b = 56 * 78;
  c = 34 + 23;
  }
```

5.4 Concatenazione di costrutti di selezione semplice

Le istruzioni A e B menzionate nel costrutto di selezione (A come istruzione da eseguire quando l'espressione e vera e B in alternativa) possono essere a loro volta gruppi di istruzioni `{.....},` oppure `if-else`, o altre istruzioni che verranno mostrate in seguito. Pertanto le istruzioni di selezione possono essere sia semplici come composte, ovvero annidate in cascata quando si succedono due o più "`if`". Per esempio, si può avere:

```
if ( <Espressione 1> )
   <Istruzione A>;
else if ( <Espressione 2> )
   <Istruzione B>;
else if ( ............... )
   .........
else
   <Istruzione Z>;
```

Il massimo numero di nidificazioni ammesse per il costrutto di selezione dipende dal compilatore utilizzato.

5.5 La soluzione di una equazione di secondo grado

In questa sessione viene riportato un programma che, dati i coefficienti a, b, c dell'equazione di secondo grado

$$ax^2 + bx + c = 0$$

calcola, se esistono, le soluzioni reali. La soluzione richiede l'analisi dei ben noti casi particolari.
In primo luogo occorre stabilire se a<=0, dato che nel caso opposto l'equazione degenera al primo grado.
Se a<=0 allora la soluzione dipende dal segno di

$$\Delta = b^2 - 4ac$$

Se delta>0 le soluzioni sono reali, altrimenti hanno componente immaginaria non nulla.

Il seguente programma traduce direttamente la precedente analisi dei casi.

```
void main(void)
{
float a, b, c, x1, x2, delta;

printf("Calcolo delle radici dell'equazione di secondo grado: \n");
printf("       ax^2 + bx + c = 0 \n");

/* inserisci i valori dei parametri a, b, e c */
printf("inserisci i parametri a, b, c, nell'ordine: \n");
scanf("%f %f %f", &a, &b, &c);

printf("Radici dell'equazione con coeff. a = %f b = %f c = %f\n", a,
b, c);

if (a != 0) /* caso di equazione non degenere */
   {  /* calcola e visualizza le radici di ax^2 + bx + c = 0 */
```

```
   delta = b*b - 4*a*c; /* calcolo del delta */
   if (delta >= 0)    /* calcolo di x1, x2 */
      { x1 = (- b - sqrt(delta))/(2*a);
      x2 = (- b + sqrt(delta))/(2*a);
      printf("Due radici reali: x1 = %f,   x2 = %f \n", x1, x2);
      }
   else  /* delta < 0 */
      printf("Nessuna radice reale \n");
   }
else if (b != 0) /* caso degenere */
   { /* coeff. del termine di primo grado diverso da zero */
   x1 = - c/b;   /* calcolo della radice     */
   printf("Una sola soluzione: x = %f \n", x1);
   }
else   /* coeff. del termine di primo grado uguale a zero */
   printf("Nessuna soluzione! \n");
}
```

Utilizzando il programma si ha:

```
Calcolo delle radici dell'equazione di secondo grado:

       ax^2 + bx + c = 0

inserisci i parametri a, b, c, nell'ordine: 1 6 5
Radici dell'equazione con coeff. a = 1.000000 b = 6.000000 c =
5.000000
Due radici reali: x1 = -5.000000,   x2 = -1.000000
```

5.6 Controllo di consistenza della data

Il seguente esempio implementa un programma per il controllo di consistenza delle date. Il programma verifica che la data inserita sia possibile. Questa verifica deve tenere conto della durata dei singoli mesi in termini di giorni e della presenza o meno dell'anno bisestile.

Un anno e' bisestile quando si hanno 29 giorni nel mese di Febbraio. Questa caratteristica non si ha generalmente negli anni divisibili per 4. Prendere per vera questa assunzione sarebbe un errore grave. In effetti, il mese di Febbraio deve avere 29 giorni solo negli anni divisibili per 4 che non sono divisibili anche per 100. Il 29/2/200 non esiste poiché nonostante sia divisibile per 4 lo anche per 100.

Fanno eccezione a questa regola gli anni che sono divisibili per 400, come il 2000 che risulta bisestile. Questo specifica condizione deve essere considerata come una parte del cosi' detto Problema del Millennio, Millennium Bug. Molti dei sistemi automatici per tenere conto della data corrente possono non avere tenuto conto che nonostante il 2000 sia divisibile per 4 lo e' anche per 100 e per 400, facendo di questo un anno bisestile e non un anno non bisestile.

```
void main (void)
```

```
{
int g, m, a, ng;
printf("Inserisci una data: \n"); /*inserimento dei dati*/
printf("giorno:");  scanf("%d", &g);
printf("mese:");    scanf("%d", &m);
printf("anno:");    scanf("%d", &a);

if (m<1 || m>12) printf("Il mese %d non e valido", m);
else
 {
 if (m == 11 || m == 4 || m == 6 || m == 9) ng = 30;
 else if (m == 2)
  {  /*controllo se l'anno e' bisestile*/
  if (((a % 4 == 0) && (a % 100 != 0)) || (a %400 == 0)) ng = 29;
  else ng = 28;
  }
 else ng = 31;
 if (g < 1 ||  g > ng) printf("Il giorno %d non e' valido", g);
 else printf("Data valida \n");
 }
}
```

5.7 Le difficoltà di comprensione e possibili problemi nei costrutti di selezione

Una prima sorgente di difficoltà nei costrutti di selezione composta e semplice consiste nella comprensione e nella specifica delle condizioni che determinano la selezione. E' noto di fatti come una stessa condizione possa essere specificata in modo diverso ma con lo stesso significato. Si faccia per esempio riferimento al teorema di De Morgan dell'algebra di Boole.

Il costrutto `if-else` può dare luogo a molte ambiguità. Per esempio, nel programma che segue, la sequenza di costrutti di selezione e' equivoca. Per un programmatore inesperto, può non essere chiaro se l'istruzione associata all'`else` viene eseguita quando `A<=2` oppure quando `B>=3`. In effetti viene eseguita quando entrambe le condizioni sono vere:

```
if (A>2) if (B<3) b=34*67;
else X=45;
```

Tali istruzioni possono essere riscritte semplicemente utilizzando un'indentazione che faciliti la lettura:

```
if (A>2)
  if (B<3) b=34*67;
  else X=45;
```

L'assegnazione `x=45` e' eseguita solo quando `A>2` e `B>=3`.

La semantica di dette istruzioni e' diversa dalla seguente soluzione:

```
if (A>2 && B<3) b=34*67;
```

```
else X=45;
```

che porta ad eseguire l'assegnazione di X quando A<=2 oppure B>=3. L'indentazione e' solo un metodo per facilitare la lettura, ma un'indentazione errata può condurre ad erronee interpretazioni. Per esempio, la seguente indentazione suggerisce un'interpretazione sbagliata:

```
if (A>2)
  if (B<3) b=34*67;
else X=45;
```

Nonostante l'indentazione il significato rimane quello dell'esempio precedente, in quanto in assenza di delimitatori il compilatore associa l'opzione else all'`if` precedente. Pertanto, se il programmatore desiderava porre l'alternativa else alla condizione `A>2` doveva utilizzare il costrutto sequenza e quindi le parentesi graffe, `{}`:

```
if (A>2)
  { if (B<3) b=34*67; }
else X=45;
```

Questo ha ovviamente un diverso significato rispetto agli esempi precedenti.

5.8 Problemi in composizione di costrutti di selezione semplice

Quando vengono definite selezioni composte da selezioni semplici, un problema frequente e' quello della scelta delle condizioni. Sorgono di solito ambiguità se, come nell'esempio che segue, le condizioni non si escludono mutualmente. Per esempio, se per `B==8` dovrebbero venire eseguite tutte le tre istruzioni, a causa della struttura di `if` in cascata, viene eseguita solo l'`<Istruzione 1>`. In tali casi, ha molta importanza la successione con la quale vengono poste le condizioni di selezione.

```
if (B>5 && B<10) <Istruzione 1>;
else if (B>6 && B<20)  <Istruzione 2>;
else if (A==2 || B==8) <Istruzione 3>;
..........
else <Istruzione N>;
```

Oltre ai problemi discussi, in C vi sono altre fonti di ambiguità che sono dovute al fatto che tale linguaggio accetta come condizione del costrutto `if-else` espressioni di qualsiasi tipo. Inoltre, nella condizione vi possono essere anche assegnazioni. Queste assegnazioni ed espressioni possono essere inserite come condizioni o semplicemente accodate alla condizione con una virgola (un altro modo per separare le istruzioni del C). In altri termini, come espressione si ha una sequenza di istruzioni semplici. Per esempio, si osservino le istruzioni di selezione riportate nel programma sottostante. Per `a==4` vengono effettuate le stampe 2 e 3, per `a==0` solo la 5, e quando `"a"` assume un qualsiasi altro valore non viene stampato niente.

```
int a,b,c;

printf("a = "); scanf("%d",&a);
```

```
if ( b=0 )       printf("1) B=%d\n",b);
if ( a==4 )      printf("2) A=%d\n",a);
if ( b=0, a==4 ) printf("3) A=%d, b=%d\n",a,b);
if ( a==4, b=0 ) printf("4) A=%d, b=%d\n",a,b);
if ( a==(b=0) )  printf("5) A=%d, b=%d\n",a,b);
```

Questo e' dovuto al fatto che, quando vi e' un'assegnazione, al posto della condizione viene considerato il valore dell'assegnazione per calcolare la condizione. Inoltre, quando si hanno più condizioni e assegnazioni concatenate da virgole viene considerata ai fini del calcolo della condizione solo l'ultima (quella più a destra). In C, questi problemi si hanno anche nelle definizioni delle condizioni dei costrutti `switch-case, do-while`, e `for`.

5.9 Il costrutto di selezione composta: `switch, default, break`

Quando ci sono varie possibili alternative si può ricorrere alla selezione composta con e senza alternativa. Questa struttura di controllo permette di individuare le istruzioni da eseguire in base a differenti valori assunti da un'espressione. La selezione composta viene introdotta dalla parola chiave `switch` seguita dall'espressione che deve essere valutata fra parentesi tonde e segue la seguente sintassi:

```
switch (<Espressione>)
  {
  case <Cost 1.1> :
  case <Cost 1.2> :
       .......
  case <Cost 1.M1>: <Istruzione 1>;
                    .......
                    break;
  case <Cost 2.1> :
  case <Cost 2.2> :
       .......
  case <Cost 2.M2>: <Istruzione 2>;
                    .......
                    break;

  ....  .......
  ....  .......
                    break;
  case <Cost N.1> :
       .......
  case <Cost N.MN>: <Istruzione N>;
                    .......
                    break;
  default         : <Istruzione Alternativa>;
                    .......
                    break;
  }
```

Fra le parentesi graffe, e' riportato l'elenco delle varie possibilità. Per esempio, se l'espressione assume il valore costante intera pari a `<Cost1.1>,` oppure `<Cost1.2>,` etc. allora viene eseguita l'`<Istruzione 1>.`

Associate alla parola chiave `default` possono essere poste le istruzioni che devono essere eseguite se l'espressione produce un valore che non e' contemplato nei vari casi. La parola chiave default può essere posta in qualsiasi posizione, anche affiancata a condizioni definite, oppure omessa. In caso di omissione, se l'espressione assume un valore non contemplato dai valori delle costanti allora non viene eseguita nessuna delle alternative proposte dal costrutto e il controllo passa all'istruzione successiva al costrutto `switch` stesso.

5.10 Confronto fra `if` concatenati e il costrutto `switch`

Si noti che lo stesso risultato che si può ottenere con il costrutto `switch` può essere ottenuto attraverso l'utilizzo di costrutti di selezione semplici nidificati:

```
if ((<Espressione>==<Cost 1.1>) || ...  ==<Cost 1.M1>  )
                              <Istruzione 1>;
else if ((<Espressione>==<Cost 2.1>) || ... ==<Cost 2.M2> )
                              <Istruzione 2>;
  ...           ...                ...              ...
else if ((<Espressione>==<Cost N.1>) || ... ==<Cost N.MN> )
                              <Istruzione N>;
else <Istruzione Alternativa>;
```

Questa forma risulta complessa da comprendere anche con poche alternative.

In seguito sono riportate le differenze fra il costrutto di selezione composta ottenuto con istruzioni `if` in cascata e il costrutto `switch`:

- Lo `switch` lavora con un semplice controllo per uguale, mentre nel costrutto `if` concatenato si possono avere anche condizioni complesse. In caso di valore duplicato relativo a due `case` dello stesso `switch` viene dato errore.

- Nello `switch` le valutazioni sono indipendenti l'una dall'altra, mentre nel costrutto `if` si accede al controllo successivo solo se il primo si è dimostrato falso.
- Lo `switch` si può considerare una sequenza, o una cascata di `if` dipendentemente dall'uso dell'istruzione `break`.

Ad ogni `case` del costrutto possono essere associate più istruzioni senza aver bisogno di raggrupparle per mezzo di parentesi graffe. In esecuzione, le espressioni costanti che individuano i vari casi vengono confrontate con il valore assunto dall'espressione. Una volta trovata la giusta alternativa vengono eseguite le istruzioni corrispondenti. L'istruzione di uscita, `break`, chiude immediatamente la selezione e passa il controllo all'istruzione successiva. Il gruppo di istruzioni associate ad ogni case può contenere anche più di un'istruzione di `break`.

5.11 Costrutto di selezione in linea del C/C++

Questo costrutto può essere direttamente inserito in espressioni e presenta la seguente sintassi espressa in BNF:

```
<Espressione> ?  <Espressione se vera> : <Espressione se falsa>
```

L'<Espressione> e' la condizione della selezione, questa può essere una qualsiasi espressione algebrica come quelle utilizzate per la condizione del costrutto di selezione semplice. Quando tale espressione produce un valore diverso da zero viene considerata vera, altrimenti falsa. Quando l'espressione e' vera viene eseguita e quindi stimata l'<Espressione se Vera> altrimenti viene eseguita e stimata l'<Espressione se falsa>. Queste due espressioni possono includere anche assegnazioni. Comunque l'intero costrutto può essere considerato una <Espressione> e quindi può essere utilizzato in espressioni algebriche. Il costrutto produrrà come risultato quello dell'espressione vera o falsa a seconda del valore dell'<Espressione>. E' in questo senso che tale costrutto viene utilizzato per la definizione delle macro per la stima del massimo e del minimo di due numeri. Si veda i punti **Errore. L'origine riferimento non è stata trovata.**, **Errore. L'origine riferimento non è stata trovata.**.

6 Modulo: Costrutti iterativi

Durante la codifica di algoritmi per mezzo di linguaggi di programmazione e' necessario poter eseguire gruppi di istruzioni in modo iterativo/ripetitivo. Si possono identificare costrutti che danno luogo a cicli definiti ed indefiniti.

Per quelli del primo tipo il numero di iterazioni e' noto a priori (si veda il costrutto `for`), mentre quelli del secondo tipo permettono di definire processi iterativi che si interrompono solo in base ad un criterio di arresto valutato al momento dell'esecuzione (si veda i costrutti `while` e `do-while`).

Pertanto si hanno i seguenti costrutti iterativi:

- `for ();` detto ciclo for
- `while ();` detto ciclo while
- `do-while ();` detto ciclo do-while

6.1 Costrutto iterativo definito, il for

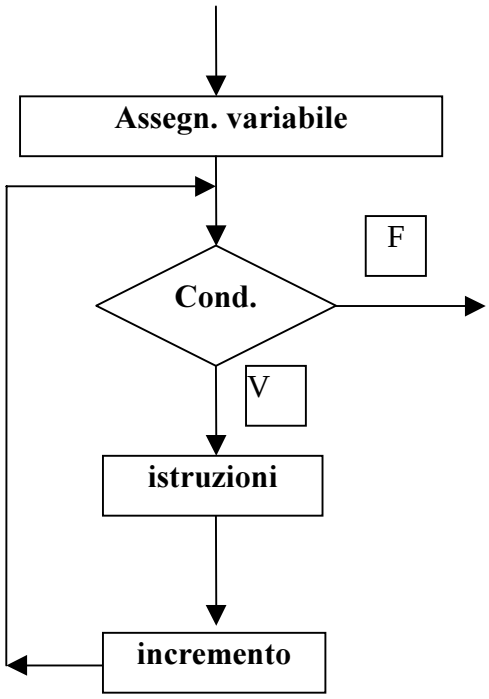
Il costrutto di iterazione per cicli definiti e' il `for`, che ha la seguente forma sintattica espressa in BNF:

```
for (<Variabile>=<Valore Iniziale>;<Condizione di Ciclo>;
     <Legge di Evoluzione>) <Istruzione>
```

L'esecuzione iterativa viene governata da una `<Variabile>` che deve essere di tipo scalare, detta indice dell'iterazione. Il ciclo inizia con l'assegnazione del valore iniziale alla variabile. Tale valore iniziale può essere assegnato anche tramite un'espressione algebrica.
Se la `<Condizione di Ciclo>` e' soddisfatta allora viene eseguita l'`<Istruzione>` interna al ciclo. Per istruzione si intende anche una sequenza di istruzioni se identificata da un coppia di parentesi graffe { }.

Dopo l'esecuzione di `<Istruzione>`, il valore della variabile può essere modificato in base alla `<Legge di Evoluzione>`, tipicamente di incremento o decremento della variabile. Tale legge può per esempio includere un'assegnazione alla `<Variabile>` per esempio con un valore addizionale di incremento. Se dopo l'aggiornamento del valore della variabile la `<Condizione di Ciclo>` risulta soddisfatta allora l'`<Istruzione>` viene eseguita nuovamente. Si noti che questa definizione e' estremamente generale. La `<Variabile>` potrebbe non essere utilizzata come riferimento nella condizione di ciclo e non essere neanche presente nella `<Legge di Evoluzione>`. Questo può portare a trasformare un ciclo iterativo in un'istruzione molto complessa e poco comprensibile.

In questo caso e' riportato il diagramma di flusso del costrutto `for`. Nella rappresentazione come diagramma di flusso si noti l'assegnazione iniziale della variabile e la verifica della condizione. La condizione viene verificata prima di eseguire la prima istruzione associata al corpo del costrutto `for`. In questo caso e' stato posto un riquadro rappresentante una legge di incremento della variabile/indice del ciclo.
In C/C++, si possono avere istruzioni qualsiasi sia al posto del blocco istruzioni che di quello incremento, come discusso in precedenza. In altri linguaggi il costrutto `for` e' molto più restrittivo, spesso non vi e' la possibilità di specificare la regola di incremento. Questa può per esempio essere solo un incremento o decremento di una unita'. E' pertanto opportuno considerare queste limitazioni quando si realizzano dei programmi che devono essere portabili, facilmente traducibili in altri linguaggi.



6.2 Esempio di uso del costrutto `for`: la stampa di un vettore di interi

Il seguente esempio riporta una porzione di codice per la stampa del contenuto di un vettore di interi:

```
#define DIMVET  6

int av[DIMVET] = { 1, 45, 34, 33, 29, 78};
……..

int i;
……..

for (i=0;i<DIMVET;i=i+1)
  printf("av[%d] = %d \n", i, av[i]);
```

In questo esempio e'stato utilizzato il simbolo `DIMVET` per la definizione di una costante, la dimensione del vettore. Tale valore e'stato utilizzato per specificare il criterio di arresto del ciclo `for`. Si noti la legge di evoluzione del ciclo iterativo. In questo caso, viene incrementato di una unita' il valore della variabile indice `i` ad ogni esecuzione della legge di evoluzione. Il valore indice i passa da 0 a 6 quando la condizione `i<DIMVET` non e'soddisfatta. Questo permette di stampare tutte le componenti del vettore `av[]`.

La legge di evoluzione può essere una qualsiasi assegnazione o anche sequenza di assegnazioni separate da virgole. Se la legge di evoluzione non provoca il cambiamento della variabile indice, e questa non e'

influenzata dall'istruzione dentro il ciclo `for`, allora il ciclo `for` viene eseguito in modo indefinito poiché il criterio di arresto non ha speranza di diventare falso:

```
for (i=0;i<DIMVET;j=i+1)
  printf("av[%d] = %d \n", i, av[i]);
```

6.3 Esempio di uso del costrutto `for`: la somma di elementi di un vettore

Un ulteriore esempio può essere la somma degli elementi di indice pari di un vettore:

```
int av[DIMVET] = { ……. } ;
int sum=0;

for (i=0;i<DIMVET; i=i+2)
   { sum = sum + av[i];
    printf("Somma parziale %d\n", sum);
   }
printf("Somma totale %d\n", sum);
```

L'uscita prodotta da questo programma e' riportata di seguito.

```
Somma parziale 21
Somma parziale 55
Somma parziale 84
Somma totale 84
```

Poiché nel costrutto di iterazione `for` e' possibile definire una qualsiasi regola di incremento per la variabile indice, la variabile indice stessa può essere anche un `float` o un carattere, `char`. Si possono inoltre realizzare cicli `for` con indice crescente, decrescente o dipendente da valori calcolati nelle istruzioni stesse eseguite in modo iterativo.

Questa ultima ipotesi e' da evitare poiché rende ostica la comprensione del funzionamento stesso del programma:

```
for (i=1;i<45;i=j+i)
  { j = i*2;
  printf ("i %d, j %d \n", i, j);
  }
```

Che produce a schermo:

```
i 1, j 2
i 3, j 6
i 9, j 18
i 27, j 54
```

6.4 Le istruzioni break e continue per i costrutti iterativi

In C, e' possibile uscire da un ciclo `for` a prescindere dalla condizione di ciclo per mezzo dell'istruzione `break`. Esiste inoltre, con l'istruzione `continue`, la possibilità di interrompere l'esecuzione dell'istruzione interna (qualora questa sia un'istruzione composta) e di saltare direttamente al punto in cui viene fatto l'aggiornamento del valore dell'indice dell'iterazione e del successivo controllo della condizione.

Per esempio si veda il seguente programma, ove l'istruzione di selezione con condizione `sum>100` attiva l'istruzione `break` che provoca l'uscita dal processo iterativo anche se questo non si e' svolto per il numero di iterazioni prefissato.

```
int av[DIMVET] = { ……. } ;
int sum=0;

for (i=0;i<DIMVET; i=i+2)
   { sum = sum + av[i];
    printf("Somma parziale %d\n", sum);
   if (sum>100) break;
   }
printf("Somma totale %d\n", sum);
```

Per ulteriori esempi si veda la sezione 7.11.

6.5 I costrutti `for` annidati

I costrutti `for` possono essere annidati. Il massimo numero di annidamenti possibili (*nesting levels*) dipende dal compilatore. In caso di cicli annidati l'esecuzione di un'istruzione di `break` comporta l'uscita dal ciclo corrente e non da tutti.

Nel seguente esempio viene calcolata la somma degli elementi di una matrice:

```
int am[DR][DC];
int i,j, sum=0;
……
for (i=0;i<DR;i=i+1)
    { for (j=0;j<DC;j=j+1)
          { sum=sum+am[i][j];
           printf("%d, ", am[i][j]);
          }
    }
printf("Somma totale %d\n", sum);
```

In questo caso, l'uscita del programma consiste nella sequenza dei dati che compongono la matrice stampati a schermo colonna per colonna. Alla fine viene stampato anche il valore totale della somma.

6.6 Ricerca del valor massimo degli elementi di una matrice

Il valore massimo di un insieme di dati e' il più grande valore assunto dalle variabili dell'insieme. Il seguente programma effettua la ricerca del valore massimo fra gli elementi di una matrice:

```
#define MINIMO -32000
……
int am[DR][DC];
int i, j, max=MINIMO, mi, mj;
……
for (i=0;i<DR;i=i+1)
  {for (j=0;j<DC;j=j+1)
     { if (max<am[i][j])
        { max = am[i][j];
        mi=i;  mj=j;
        }
     }
  }
printf("Il massimmo %d locato in (%d,%d)\n", max, mi,mj);
```

Il programma itera su tutti gli elementi della matrice. Per ogni elemento effettua il confronto con il valore della variabile `max`. Questa viene inizializzata al valore minimo della dinamica, in base al tipo di variabile utilizzato. Nell'esempio e' stato imposto a `-32000`. Ogni volta si riscontra che `max<am[i][j]` il valore di `max` viene aggiornato e vengono memorizzate anche le coordinate della componente. In questo modo, il valore e le coordinate dell'elemento di valore massimo sono memorizzate e possono essere utilizzati in seguito.

6.7 Calcolo del valor medio degli elementi di una matrice

Il valor medio degli elementi di un insieme di dati viene calcolato facendo la somma degli elementi e dividendo il risultato per il numero stesso degli elementi. Il seguente programma effettua il calcolo del valor medio degli elementi di una matrice:

```
#define MINIMO -32000
……
int am[DR][DC], i, j, sum=0;
float med;
……
for (i=0;i<DR;i=i+1)
  {for (j=0;j<DC;j=j+1)
     sum = sum + am[i][j];
  }
med = sum/(DR*DC);
printf("Valor medio %f\n", med);
```

Il programma itera su tutti gli elementi della matrice per calcolare la somma degli elementi. Dopo calcolata la somma calcola il valor medio dividendo il valore della somma per il numero degli elementi. Per ottenere una valor medio preciso e' stata utilizzata una variabile di tipo `float`.

6.8 Problemi dei costrutti iterativi

I costrutti iterativi possono dare luogo a cicli indefiniti. Il linguaggio C e' particolarmente sensibile a questi problemi poiché permette di utilizzare come condizioni di selezione espressioni di tipo qualsiasi ed anche assegnazioni. Per esempio, entrambi i programmi seguenti danno luogo a cicli indefiniti. Il primo perché la condizione di aggiornamento della variabile indice non provoca nessun incremento anche se sintatticamente corretta. Il secondo, poiché l'ultima espressione della lista contenuta nel campo condizione del `for` risulta sempre vera.

```
int i;
for (i=0;i<10;i) printf("%d, ",i);
.......
for (i=0;i<10000, printf("nulla %d\n",i), i+1; i=i+1);
```

In generale, nei cicli, siano essi definiti che indefiniti, si deve fare molta attenzione al tipo di variabili che vengono utilizzate come indice. Per esempio, nel programma che segue e' stata utilizzata come valore limite del ciclo una variabile intera che, prima del ciclo, ha assunto in qualche modo un valore maggiore del massimo rappresentabile, cioè vi e' stato un traboccamento (il compilatore non e' in grado di segnalarlo e non viene individuato neanche al momento dell'esecuzione). L'effetto, di tale operazione e' che il ciclo non viene eseguito poiché la condizione di arresto e' subito soddisfatta.

Se si ipotizza di avere un calcolatore e un sistema operativo che usano interi a 16 bit, 2 byte, con l'assegnazione `N=40000`, a causa della modalità di rappresentazione dei numeri interi nella rappresentazione in complemento, si e' assegnato a `N` un valore minore di zero e quindi la condizione di ciclo non e' mai soddisfatta. Si consideri infatti che con 16 bit il massimo numero intero con segno rappresentabile e' 2^15-1 = 32767.

```
int N,i;
N=40000;
........
for (i=0;i<=N;i++) printf("%d ",i);
```

Un altro esempio di uso inadeguato dei tipi per la definizione dei cicli `for` e' riportato nel programma che segue. Si osservi attentamente il programma alla ricerca di un errore logico nella programmazione. L'effetto dell'esecuzione di questo programma e la realizzazione di un ciclo indefinito. Questo e dovuto al fatto che l'indice i ha come condizione di fine ciclo il valore oltre il quale si ha traboccamento della variabile definita come `unsigned char`, che e rappresentato con un singolo byte. Il problema non può essere eliminato cambiando la condizione di arresto (per esempio ponendo `i==255`), ma deve essere utilizzato un tipo di variabile che possa rappresentare anche il 256. Alternativamente si possono far fare al ciclo solo 255 iterazioni da 0 a 254 e l'ultima eseguirla fuori dal ciclo, oppure utilizzare un ciclo `while` o `do-while` con opportuni accorgimenti.

```
unsigned char i;
```

```
........
for (i=0;i<=255;i++) printf("%d ",i);
```

6.9 Calcolo dei numeri primi

Con il seguente programma di esempio e' possibile calcolare in modo iterativo tutti i numeri primi a partire da 1 fino ad un numero indicato. Il procedimento si basa direttamente sulla definizione di numero primo.

Un numero e' un numero primo se può essere diviso con restozero solo per 1 e per se stesso.

Il meccanismo iterativo utilizzato si basa su questa definizione e praticamente prova a dividere ogni numero per tutti i numeri che sono la meta' di lui stesso e per ogni numero inferiore a tale valore.

```
void main(void)
{
int n, i, j, flag;

printf("Ricerca di tutti i numeri pari fino a: ");
scanf("%d", &n);

for (i=2; i<=n; i++) /*per ogni numero da 2 fino a quello massimo*/
 { flag = 1;
 for (j=2; j<=i-1; j++) /*ricerca dei divisori del numero*/
  { if ((i % j) == 0)   /*verifica se e' divisibile*/
   { flag = 0;         /* e' divisibile per 2, non e' pari*/
   break;
   }
  }
  if (flag) printf("%d ", i);
 }
}
```

Esempio di esecuzione:

```
Ricerca di tutti i numeri pari fino a: 100

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

6.10 Costrutto GOTO

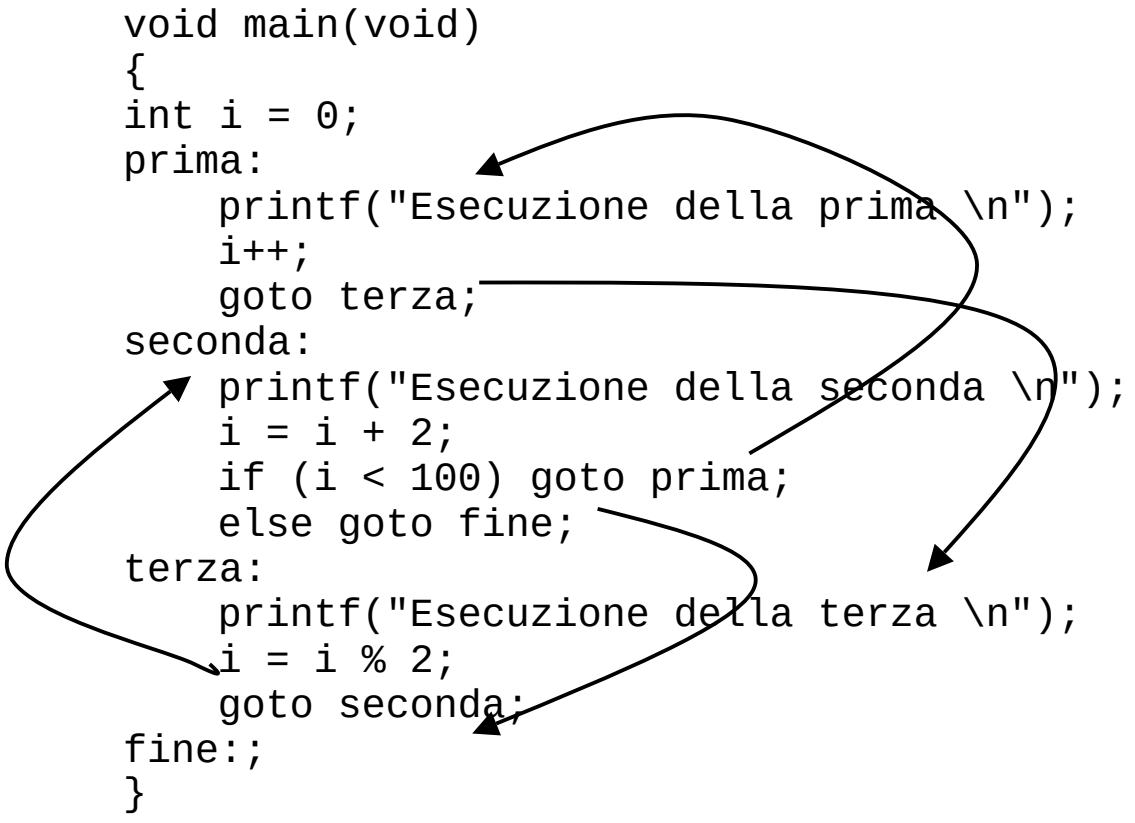
Il costrutto GOTO permette di effettuare un salto incondizionato come il costrutto JMP dei linguaggio assemblativi. Con l'uso del costrutto di selezione si possono realizzare dei costrutti di salto condizionato anche complessi.

```
saltaqui: ;  // queste e' una etichetta, cioe' una label
```

```
……..
saltala: ;   // questa e' una etichetta cioe' una label
……..
goto saltaqui;  // salto incondizionato
……
…….
if (a>34) goto saltala:; // salto condizionato
```

Nei linguaggi di alto livello il costrutto GOTO deve essere utilizzato con estrema attenzione. Si consiglia di non utilizzarlo o di utilizzarlo solo in casi eccezionali. Nell'uso corrente della realizzazione di algoritmi ed elaborati del corso di fondamenti di informatica e' del tutto improbabile che vi sia la necessita' di utilizzarlo. Ogni successione di istruzioni di selezione ed iterazioni può essere realizzata semplicemente facendo uso dei precedenti costrutti di selezione ed iterazione. In generale, ogni algoritmo può essere convertito in una sequenza di istruzioni di tipo sequenza, selezione, ed iterazione.

Il costrutto GOTO può rendere difficile la lettura di un programma. Si veda il seguente esempio che, contrariamente a quanto uno può dedurre dopo la prima occhiata, instaura un ciclo iterativo che non porta alla conclusione del programma, il programma non gode della proprietà di finitezza.



```
void main(void)
{
int i = 0;
prima:
    printf("Esecuzione della prima \n");
    i++;
    goto terza;
seconda:
    printf("Esecuzione della seconda \n");
    i = i + 2;
    if (i < 100) goto prima;
    else goto fine;
terza:
    printf("Esecuzione della terza \n");
    i = i % 2;
    goto seconda;
fine:;
}
```

7 Modulo: Costrutti di iterazione indefinita

In questo modulo sono presentati i costrutti di iterazione indefinita del C/C++. Oltre a indurre i costrutti questo modulo presenta anche alcune loro applicazione in forma di esempi ed esercizi.
E' assolutamente necessario avere compreso i precedenti moduli prima di procedere con questo.

In C vi sono due tipi di costrutti iterativi che possono produrre cicli indefiniti. Questi si distinguono per la posizione in cui viene effettuato il controllo della condizione di arresto. Infatti, tale controllo può essere effettuato in ``testa" (come per il costrutto `for`), oppure in "coda" (quando il costrutto permette l'esecuzione dell'istruzione associata per almeno una volta a prescindere dalla condizione di arresto).

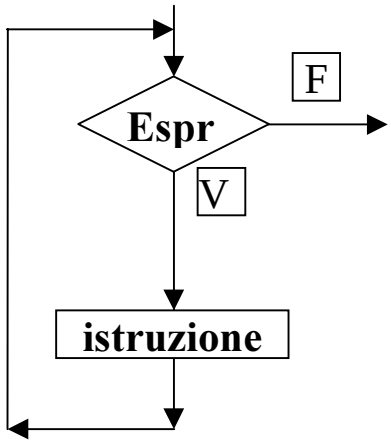
Sono costrutti iterativi con controllo in testa il costrutto `while` del Pascal e C, mentre sono costrutti iterativi con controllo in coda i costrutti `do-while` del C e il `repeat-until` del Pascal.

7.1 Il Costrutto iterativo: `while`

I cicli indefiniti, con il controllo in testa, si possono costruire per mezzo della seguente istruzione:

```
while (<Espressione>) <Istruzione>;
```

L'esecuzione di questa istruzione comporta la valutazione dell'`<Espressione>`. Se tale espressione risulta vera (diversa da zero) allora viene eseguita l'`<Istruzione>`, altrimenti il controllo passa all'istruzione successiva all'istruzione `while` stessa. Dopo l'esecuzione dell'istruzione il ciclo ritorna a valutare l'`<Espressione>` per eseguirla finche' questa risulta vera. Il programma riportato di seguito e' in grado di produrre tutti i numeri divisibili per 13, presenti nell'intervallo da 0 a 13000:




```
float a=0;
while (a<=13000)
   { if (a%13==0) printf ("%f \n", a);
   a = a + 1;
   }
```

In questo caso, lo stesso programma poteva essere realizzato agevolmente con un costrutto `for`.
Anche da questo costrutto iterativo e' possibile uscire anche se l'`<Espressione>` non e' falsa, cioè prematuramente rispetto a quanto pianificato.

Questo e' possibile utilizzando l'istruzione `break`, per esempio il ciclo viene interrotto nel seguente programma quando `a==b`.

```
float a=0;
while (a<=13000)
   { if (a%13==0) printf ("%f \n", a);
   if (a==b) break;
   a = a + 1;
   }
```

Con la parola chiave `continue` si può interrompere il ciclo e saltare direttamente in testa al ciclo dove viene valutata l'`<Espressione>.`

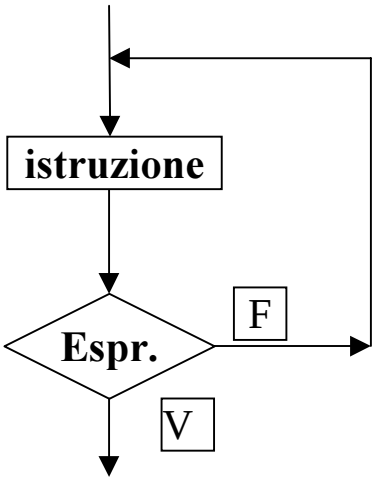
Dato un certo programma codificato utilizzando un costrutto `for` e' sempre possibile riscriverlo utilizzando il costrutto `while`.

E' importante pertanto mettere in chiaro quando e' opportuno utilizzare tali costrutti. In generale, il costrutto `for` dovrebbe essere utilizzato quando il numero di iterazioni e' conosciuto a priori mentre il ciclo `while` quando tale numero non e' conosciuto a priori ma dipende direttamente dalle istruzioni che sono contenute nel corpo stesso dell'istruzione.

7.2 Il Costrutto iterativo: `do-while`

Il duale del ciclo `while` e' il ciclo `do-while` che presenta la seguente sintassi con il controllo per l'iterazione in "coda":

```
do <Istruzione> while <Espressione>;
```



L'esecuzione di questa istruzione comporta l'esecuzione dell'<Istruzione> interna alle parole chiave `do-while` prima della valutazione dell'`<Espressione>.` Se tale `<Espressione>` risulta falsa allora le istruzioni interne vengono eseguite nuovamente, altrimenti si esce dal costrutto iterativo e viene eseguita l'istruzione successiva all'intera istruzione. Pertanto, con tale costrutto le istruzioni contenute vengono eseguite comunque almeno una volta. Inoltre il valore dell'`<Espressione>` viene controllato in coda differentemente dal costrutto `while` dove il controllo era effettuato in testa.

Si noti che anche in questo caso l'`<Espressione>` viene considerata vera quando assume un valore diverso da zero.

Il programma riportato di seguito e' in grado di produrre una sequenza di numeri prodotti da una certa funzione `f()` e di selezionare fra questi solo quelli pari:

```
float a=0, val;
do { val = f(a);
   if (val%2==0) printf ("val=%f, con a=%f\n", val, a);
```

```
   a = a + val;
   } while (val<=13000);
```

In questo caso, lo stesso programma poteva essere realizzato con un costrutto `for` utilizzando il costrutto `break` per uscire quando la condizione sul valore di `val` e' soddisfatta se si conosce a priori il numero di iterazioni che al massimo portano a completare la stampa della successione di numeri. Anche da questo costrutto iterativo e' possibile uscirne a prescindere dal criterio definito con l'`<Espressione>` utilizzando l'istruzione `break`, mentre con l'istruzione `continue` si passa direttamente a valutare l'`<Espressione>`.

Dato un certo programma codificato utilizzando un costrutto `for` e' sempre possibile riscriverlo utilizzando il costrutto `do-while`, anche se molto spesso non e' conveniente a causa della differenza di posizione dell'espressione di controllo. Il `for` effettua un controllo in testa mentre il `do-while` lo effettua in coda. Pertanto trasformare un ciclo `for` in un ciclo `do-while` comporta spesso fare in modo che anche alla prima iterazione i due cicli si comportino nello stesso modo.

7.3 Cicli indefiniti, senza controllo esplicito

Si possono realizzare anche cicli indefiniti senza possibilità di uscita (interruzione del ciclo) in base alla condizione di arresto. In C questi cicli possono essere realizzati nel modo seguente:

```
for(;;)
  { <Istruzione>;
  }

while (1)
  { <Istruzione>;
  }

do {
   <Istruzione>;
   } while(1);
```

E' facilmente comprensibile come il ciclo iterativo possa procedere in modo indefinito senza possibilità di interrompersi, infatti l'`<Espressione>` di controllo non e' presente oppure e' sempre vera.

Comunque e' possibile uscire da tali cicli iterativi per mezzo dell'istruzione `break`. A questo riguardo si veda la sezione 6.4.

7.4 Il passo generico e i criteri di arresto

Quando si ha un procedimento iterativo indefinito si devono tenere conto di due aspetti: il passo generico ed il criterio di arresto. Ogni procedimento iterativo può essere formalizzato evidenziando oltre al passo

generico anche i procedimenti iterativi non indefiniti. Il passo generico esula dai dettagli della prima e dell'ultima iterazione. Si veda per esempio il seguente problema.

Disposizioni con ripetizione
Scrivere una procedura per generare le disposizioni con ripetizione di N elementi a classe K.

E' facile verificare che il numero delle disposizioni con ripetizione di N elementi a classe K, e' pari a N alla K. La proprietà vale per K=1. Supporla vera per K-1 significa che il numero degli elementi in quel caso sarebbe N alla K-1. Per creare le disposizioni con ripetizione di N elementi a classe K, a partire dalle disposizioni dei soliti N elementi a classe K-1, e' sufficiente comporre K-uple dove ciascuna ha al primo posto uno degli N elementi da disporre. Si hanno pertanto N gruppi di disposizioni, ciascuna di N alla K-1 elementi. Si consideri il seguente esempio:

Disposizione degli interi 1,2,3 a classe 2. Le disposizioni risultano:

(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)

Proponiamo un semplice algoritmo iterativo basato sul concetto di numerazione con sistemi posizionali. Per generare le disposizioni dell'esempio precedente e' infatti sufficiente convertire i numeri I={ i: 0<= i <= 8 }, in base 3, essendo D_2^3=9, cioè la base corrispondente alla classe delle disposizioni. Cosi' facendo, la conversione di tali numeri conduce a creare l'insieme:

(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)

Per ottenere le disposizioni con ripetizione desiderate e' allora sufficiente sommare ``1" a tutti gli elementi. Il problema principale e' dunque quello di convertire il generico i di I nella base corrispondente alla classe delle disposizioni. L'algoritmo di soluzione consiste dunque in un ciclo, con un numero di iterazioni pari a N alla K, all'interno del quale si esegue, su K cifre, la conversione del generico indice in base K. Si noti che la conversione di i in I avviene sempre su K bit e che pertanto il metodo delle divisioni successive utilizzato per la conversione di base non usa l'ordinario criterio di arresto basato sull'annullamento del quoziente della divisione.

```
ConversioneBase(int X, int B, int K, int V[])
{ int I,J;  for (J=0;J<K;J++)  { V[K-J-1]= X % B;      X= X/B; } }

DisposizioniRipetizione1(int N, int K)
{ int V[10], I, J;
for (I=0;I<=(int)pow((float)N,K)-1;I++)
   { ConversioneBase(I,N,K,V);
   for (J=0;J<K;J++) printf("%d ",1+V[J]);
   printf("\n");
   }
}

main(void)
{ int X,Y;
printf("\nCalcolo delle disposizioni con ripetizione\n");
printf("elementi "); scanf("%d",&X);
```

```
printf("a classe "); scanf("%d",&Y);
DisposizioniRipetizione1(X,Y);
}
```

Utilizzando le procedure per il calcolo della potenza e per la conversione di base, la procedura DisposizioniRipetizione1 appare molto semplice. Assumendo che la variabile i del ciclo nella procedura ‘‘DisposizioniRipetizione1” sia un intero, il suo massimo valore e’ 32.767, il che limita pesantemente il campo di applicazione della procedura. Si potrebbe parzialmente limitare questo problema utilizzando per i il tipo `longint`; in tal caso, essendo il numero codificato su 32 bit con segno, si rappresentano numeri fino a 2 alla 32 pari a 2.147.483.647. Il problema e’ tuttavia solo rimandato, ma non risolto. In effetti l’origine di tale limitazione risiede nel calcolo di D_k^n che, essendo un esponenziale, ‘‘mette in crisi” ogni rappresentazione preconfezionata con linguaggi di programmazione. Una possibile soluzione a questo problema e’ quella di utilizzare delle rappresentazioni dedicate per tale numero, in modo da superare eventualmente anche la barriera dei `longint`. Naturalmente, in linea di principio, la caratteristica di questo algoritmo e’ quella di esplodere esponenzialmente in termini di spazio. Quando il numero delle disposizioni con ripetizione supera il massimo valore rappresentabile nell’elaboratore, il programma si arresta senza eseguire il ciclo, dato che il calcolo della potenza produce, per via dell’errore di rappresentazione, un valore negativo per la potenza N alla K. Questo problema si può risolvere considerando che il calcolo di N alla K potrebbe essere evitato, dato che la condizione di arresto per la creazione di disposizioni con ripetizione e' comunque individuabile controllando l'ultima disposizione. La seguente procedura mostra tale alternativa:

```
DisposizioniRipetizione2(int N, int K)
{ int I=0,J,NumCon,CifraMassima,ValoreMassimo,V[MasNum];

do { NumCon=I;
   ValoreMassimo=TRUE;
   for (J=1;J<=K;J++)
      { V[K-J]= NumCon % N; NumCon= NumCon/N;
      if (V[K-J] == N-1) CifraMassima= TRUE; else CifraMassima= FALSE;
      ValoreMassimo= ValoreMassimo & CifraMassima;
      }
   for (J=0;J<K;J++) printf("%d ",V[J]+1);
   printf("\n");
   I++;
   } while (! ValoreMassimo);
}
```

L'idea di base di questa procedura e’ sempre la stessa di ``DisposizioneRipetizione1". La differenza fondamentale risiede appunto nel controllo di fine ciclo, questa volta gestito tramite il costrutto `do-while`. Per problemi tali che D_k^n = n^k supera la rappresentazione usata per le variabili intere, questa procedura inizialmente funziona correttamente, fino a quando ``NumCon", che e' il numero da convertire, raggiunge il valore massimo rappresentabile.
A tal punto la procedura ovviamente crea disposizioni errate, ripetendo quelle già generate. In effetti, l'esplosione spaziale dipende dal problema in oggetto. Si noti che tale soluzione non ha bisogno delle procedure per il calcolo della potenza.

7.5 Calcolo della Radice quadrata

Nel seguente esempio viene proposto un algoritmo per il calcolo della radice quadrata di un numero per via iterativa. Il procedimento procede per approssimazioni successive ed arriva alla determinazione della radice quadrata del numero. Il passo base dell'algoritmo specifica una relazione fra il valore della radice all'iterazione n, iter, e il valore all'iterazione n+1, iter1:

$$Iter_{n+1} = \frac{Iter_n + n / Iter_n}{2}$$

Al passo zero Iter viene posto ad 1.

7.6 Programma per il calcolo della radice quadrata

```
void main(void)
{
int n, numiter = 0;
double iter, iter1;
printf("Inserisci il numero di cui calcolare la radice: ");
scanf("%d", &n);
iter = iter1 = 1;
do
 { iter = iter1;
   iter1 = 0.5*(iter+(n/iter));
   numiter++;
   printf("Iterazione %d: %.15e \n", numiter, iter1);
 }
 while (iter != iter1);
 printf("\nLa radice di %d e' %.15e \n", n, iter);
}
```

7.7 Prove di Esecuzione: calcolo della radice quadrata

Di seguito e' riportata la traccia di esecuzione del programma per il calcolo della radice quadrata:

```
Inserisci il numero di cui calcolare la radice: 26
Iterazione 1: 1.350000000000000e+001
Iterazione 2: 7.712962962962963e+000
Iterazione 3: 5.541955671157352e+000
Iterazione 4: 5.116720163987656e+000
Iterazione 5: 5.099050130180595e+000
Iterazione 6: 5.099019513684702e+000
```

```
Iterazione 7: 5.099019513592785e+000
Iterazione 8: 5.099019513592785e+000

La radice di 26 è 5.099019513592785e+000
```

Naturalmente il procedimento e' in grado di stimare la radice anche quando si tratta di determinare la radice di un numero intero che sia essa stessa un numero intero.

```
Inserisci il numero di cui calcolare la radice: 2
Iterazione 1: 1.500000000000000e+000
Iterazione 2: 1.416666666666667e+000
Iterazione 3: 1.414215686274510e+000
Iterazione 4: 1.414213562374690e+000
Iterazione 5: 1.414213562373095e+000
Iterazione 6: 1.414213562373095e+000

La radice di 2 è 1.414213562373095e+000
```

7.8 Confronto fra for, while e do-while

Di seguito sono confrontate tre versioni di uno stesso programma (uno stralcio del programma principale) che permettono di produrre i numeri compresi fra 1 e 100: la prima utilizza il ciclo `for`, la seconda il ciclo `while` e l'ultima il ciclo `do-while`.

1) "for"

```
for (i=0; i<100; i++)
<Istruzioni>



(i<100);
```

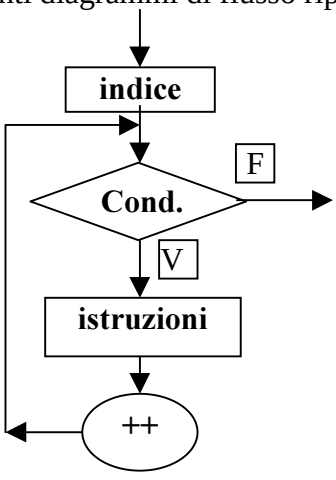
2) "while"

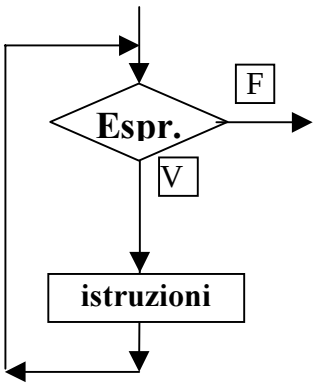
```
i=0;
while (i<100)
  {
  <Istruzioni>;
  i++;
 }
```

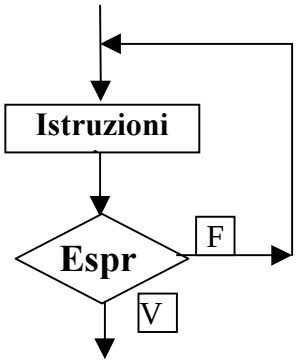
3) "do-while"

```
i=0;
do
  {
  <Istruzioni>;
  i++;
  }while
```

I seguenti diagrammi di flusso riportano il funzionamento dei rispettivi spezzoni di codice.







Ciclo `for`: Le `<istruzioni>` sono eseguite per `i=0`, l'indice viene incrementato e contemporaneamente l'esecuzione dell'istruzione continua finché il risultato dell'espressione risulta vero. Si ha l'uscita dopo l'ultimo controllo, quando `i=100`.

Ciclo `while`: Si esegue il primo controllo dell'espressione per `i=0` e quindi si esegue la prima istruzione. In base al valore dell'espressione si esegue le `<istruzioni>` finché questa è vera. Si esce dal ciclo dopo l'ultimo controllo, quando `i=100`, quindi la condizione non risulta più verificata.

Ciclo `do-while`: Si esegue le `<istruzioni>` interne al ciclo la prima volta con `i=0` (senza che venga fatto un controllo sull'indice o altre condizioni). Dopo il primo ciclo si esegue il controllo (questo modo di verificare la condizione di ciclo viene detto *controllo in coda*). L'uscita dal ciclo viene effettuata quando la condizione risulta falsa, quindi quando `i=100`.

E' sempre possibile trasformare un ciclo `for` (o `do-while`) in `while`, mentre non è vero il contrario in modo semplice (e' possibile se si utilizzano anche istruzioni di selezione). Se all'inizio del ciclo non è noto il numero di iterazioni necessarie, allora non e' consigliabile utilizzare il ciclo `for`. In base a queste considerazioni è lecito dire che il ciclo `while` è più flessibile, perché può essere sempre utilizzato al posto degli altri costrutti di iterazione, ovviamente ad un certo costo in termini del numero di istruzioni. Se all'inizio del ciclo è noto il numero di iterazioni da fare, è conveniente utilizzare l'istruzione `for`, in quanto risulta più semplice da utilizzare e più facile da leggere.

7.9 Il gioco dei fiammiferi

In questa sessione viene riportato un esercizio relativo alla realizzazione di un programma per il gioco dei fiammiferi. Tale gioco si basa su semplici regole.
Ci sono sul tavolo 11 fiammiferi. Due giocatori, alternativamente, ne raccolgono da 1 a 3 alla volta fino a esaurirli. Perde chi raccoglie l'ultimo fiammifero.

Il problema si divide sostanzialmente in due casi a seconda che il calcolatore giochi per primo o per secondo. In questi due casi si devono adottare due diverse strategie.

La prima mossa al calcolatore

E' facile dimostrare che esiste una strategia vincente per chi ha la prima mossa.
Si consideri la strategia di gioco del seguente programma
Nella variabile `numf` e' contenuto il numero dei fiammiferi sul tavolo, mentre in `na` il numero dei fiammiferi tolti dall'utente ed in `nv` quello che il programma seleziona.

```
void main(void)
{
int numf,nv,na;
nv=0;
printf("Prendo 2 fiammiferi\n");
numf=9;   /* pertanto si parte da 9 fiammiferi */
```

```
do{       /* ripeto finche' non rimane che un fiammifero */
  do{     /* ripeto finche' non vengono tolti i fiammiferi */
    printf("Quanti fiammiferi prendi? ");
    scanf("%d",&na);
    if (na<1 || na>3) printf("Non barare ...\n");
    } while (na<=0 || na>=4);
  nv=4-na;              /* strategia fissa per il computer */
  printf("Prendo %d fiammiferi\n",nv);
  numf+=-na-nv;      /* aggiornamento del numero dei fiammiferi */
  } while (numf!=1);
printf("Ho vinto!!!\n");
}
```

La strategia del programma e' manifestamente vincente. Al primo passo toglie 2 fiammiferi, lasciando sul tavolo 9 fiammiferi. Successivamente, la strategia e' quella di prendere un numero di fiammiferi pari al complemento a 4 di quelli presi dall'avversario. Poiché 9 = 4*2 + 1, dopo 2 iterazioni viene lasciato un solo fiammifero all'avversario. Ovviamente, il gioco si estende al
caso di un numero di fiammiferi pari a 2 + 4k, per k intero; ovvero si ha lo stesso risultato con 15, 19, …. fiammiferi.

La prima mossa all'utente

Se l'utente conosce la strategia vincente e la impiega senza errori l'elaboratore non puo' che perdere. Se tuttavia commette almeno un errore, allora e' facile individuare una strategia vincente. Occorre individuare l'errore e selezionare un numero di fiammiferi tale da ripristinare il caso in cui l'elaboratore gioca per primo. Si ha ovviamente che

- dopo il rilievo dell'errore l'utente sbaglia se, dopo che ha giocato, il numero dei fiammiferi rimasti sul tavolo nf e' tale che (nf-1) % 4 != 0;

- nel caso in cui l'elaboratore ha la prima mossa il ripristino avviene prendendo un numero di fiammiferi `nv` pari a `(nf-1) % 4.`

Nel caso in cui l'utente gioca secondo la strategia vincente, una possibile scelta e' quella di fare in modo che il gioco si protragga il piu' a lungo possibile, ovvero di prendere 1 fiammifero.
Il seguente programma implementa pertanto una strategia ottima, nel senso che vince appena l'utente commette un errore.

```
int numf,nv,na;
void main(void)
{
numf=11; nv=na=0;
do{
  do{
    printf("Quanti fiammiferi prendi? ");
    scanf("%d",&na);
    if (na<1 || na>3) printf("Non barare ...\n");
```

```
    } while (na<=0 || na>=4);
  numf-=na;
  /* attesa dello sbaglio ed eventuale correzione */
  if ( (numf-1) % 4 == 0 ) nv=1;  /* non ha sbagliato ! */
  else
    { nv=(numf-1) % 4;          /* approfitto dell'errore */
    printf("Ha! Ha!\n");
    }
  printf("Ho preso %d fiammiferi\n",nv);
  numf-=nv;
  } while (numf!=0 && numf!=1);
if (numf==1) printf("Ho vinto!!!\n"); else printf("Ho perso\n");
}
```

Di seguito e' riportata una prova di esecuzione del programma (versione dell'algoritmo in cui e' l'utente a fare la prima mossa) dove l'elaboratore sfrutta a suo vantaggio l'errore commesso dall'utente.

```
Quanti fiammiferi prendi? 2
Ho preso 1 fiammiferi
Quanti fiammiferi prendi? 1
Ha! Ha!
Ho preso 2 fiammiferi
Quanti fiammiferi prendi? 3
Ha! Ha!
Ho preso 1 fiammiferi
Ho vinto!!!
```

Si noti a tal proposito l'errore di forma nella presentazione del messaggio in cui l'elaboratore annuncia di aver preso 1 fiammifero.

7.10 Calcolo della precisione di macchina

Nella parte di rappresentazione dei dati e' stata mostrata la rappresentazione in virgola mobile. Tale rappresentazione presenta una precisione finita oltre la quale non e' possibile andare. Numeri piu' piccoli della precisione della rappresentazione sono considerati pari a 0 per il calcolatore. Ogni programmatore deve conoscere i limiti di rappresentazione del proprio calcolatore e quindi anche la precisione di questo. Tale limite può essere noto conoscendo la forma della rappresentazione utilizzata oppure attraverso l'uso di un semplice programma di calcolo. Tale programma si basa su di un processo iterativo che divide un numero N per 2, 4, 8, 16, .. fino ad arrivare ad un numero che nonostante sia diviso non può essere distinto dal suo predecessore non diviso.

```
void main(void)
{
double eps1, eps=1;

while (eps>0)
```

```
 {  eps1=eps;
    eps /= 2; /* equivalente a eps= eps/2*/
 }
 printf("Lo zero di macchina e: %.15e \n", eps1);

/* La notazione %.15e approssima il numero cercato a 15 cifre dopo la
virgola*/
}
```

In seguito sono riportate due tracce di esecuzione del programma. La prima utilizzando numeri in virgola mobile di tipo `double` mentre la seconda di tipo `float`. I primi presentano una mantissa con un considerevole numero di bit. Il codice riportato sopra corrisponde alla versione basata sui `double`. La versione con i `float` può essere ottenuta semplicemente.

Nel caso di `double`:

```
Lo zero di macchina è: 4.940656458412465e-324
```

Nel caso di `float`:

```
Lo zero di macchina è: 1.401298464324817e-045
```

7.11 Istruzioni break e continue nei costrutti iterativi

Ogni ciclo iterativo può essere interrotto anche nel corso dell'esecuzione con due diverse modalità, queste corrispondono a due specifiche istruzioni:
break:

- interrompe prematuramente il ciclo in esecuzione; se si hanno dei cicli annidati comporta l'uscita dal ciclo in cui si esegue ma non da tutti.

***continue*:**

- nel `for` interrompe l'esecuzione del ciclo e salta al punto in cui il valore dell'indice dell'iterazione viene aggiornato;
- nel `while` interrompe l'esecuzione del ciclo e salta direttamente in testa, dove viene valutata l'espressione;
- nel `do-while` interrompe l'esecuzione del ciclo e passa a valutare l'espressione.

Se si hanno piu' cicli annidati il `continue` viene applicato solo al ciclo in cui viene eseguito.

7.12 Le Istruzioni del linguaggio di programmazione

Le possibili proposizioni dei linguaggi di programmazione possono essere divise in definizioni, dichiarazioni e istruzioni.

Le definizioni sono predicati che permettono di definire nuovi tipi di dati o simboli che possono essere utilizzati nel prosieguo del programma. I nuovi tipi di dati sono utilizzati per dichiarare costanti e variabili, mentre i simboli per effettuare sostituzioni simboliche all'interno del programma stesso (si veda l'utilizzo del costrutto `#define` del C/C++).

Le dichiarazioni sono state ampiamente descritte nelle prime sessioni di questo corso. Per dichiarazione si intende un predicato con il quale viene dichiarata una variabile associandogli uno specifico tipo. I tipi possono essere quelli di base oppure altri definiti con delle definizioni nel programma stesso.

Le istruzioni sono costrutti che permettono di codificare gli algoritmi per mezzo di operazioni elementari. Queste sono tipicamente costrutti di sequenza, selezione, iterazione ed assegnazione ma vi sono anche altre istruzioni che permettono di produrre risultati verso l'esterno e di acquisirne (istruzioni di ingresso/uscita):

```
<Istruzione> := <Sequenza> | <Assegnazione> |
                <Ingresso/Uscita> |
                <Selezione> | <Iterazione> | <label> | <goto> |
                <continue> | <return> | ……
```

Il costrutto sequenza viene realizzato in C/C++ per mezzo della coppia di parentesi `{}`. Il costrutto sequenza permette di raggruppare un insieme di istruzioni che vengono eseguite in sequenza e manipolarle come una singola istruzione. Le istruzioni singole più semplici sono quelle di assegnazione come precedentemente mostrato.

Le istruzioni di ingresso uscita permetto di acquisire dall'esterno dei dati e di produrli all'esterno (il `printf()` e' la classica istruzione di uscita, lo `scanf()` di ingresso). In questa versione del linguaggio C/C++ didattico utilizzato sono utilizzate tali funzioni per coprire le istruzioni di ingresso uscita perché risultano di più facile comprensione rispetto a quelle del C++.

La selezione può essere semplice o composta. In C/C++ si ha il costrutto `if` come selezione semplice e il costrutto `switch` come sequenza composta:

```
<Selezione> := <Istruzione If> | <Istruzione Switch>
```

Per quanto riguarda l'iterazione si hanno tre possibili costrutti:

```
<Iterazione> := <Istruzione For> |
                <Istruzione While> |
                <Istruzione Do-While>
```

7.13 La formattazione del codice, l'indentazione

L'indentazione e' un modo di formattare le istruzioni delle quali sono composti i programmi per dare risalto alle strutture di selezione ed iterazione. In tale modo e' possibile identificare con più facilita' il livello di annidamento di ogni singola istruzione, identificando quali sono le strutture di selezione ed

iterazione che la governano. Negli esempi precedenti sono state utilizzate semplici regole di indentazione per le quali ogni volta che viene inserito un costrutto sequenza, o si ha una istruzione di selezione o iterazione le istruzioni selezionate o iterate vengono spostate a sinistra delle pagina di un numero fissato di caratteri o di una tabulazione. Per esempio il seguente codice risulta correttamente indentato:

```
float a=0, val;
do{
  val = f(a);
  if (val%2==0)
    printf ("val=%f, con a=%f\n", val, a);
  for (i=0;i<DR;i=i+1)
    {
    for (j=0;j<DC;j=j+1)
      sum = sum + am[i][j];
    }
  a = a + val;
  } while (val<=13000);
```

Mentre lo stesso codice potrebbe risultare meno comprensibile con una diversa indentazione:

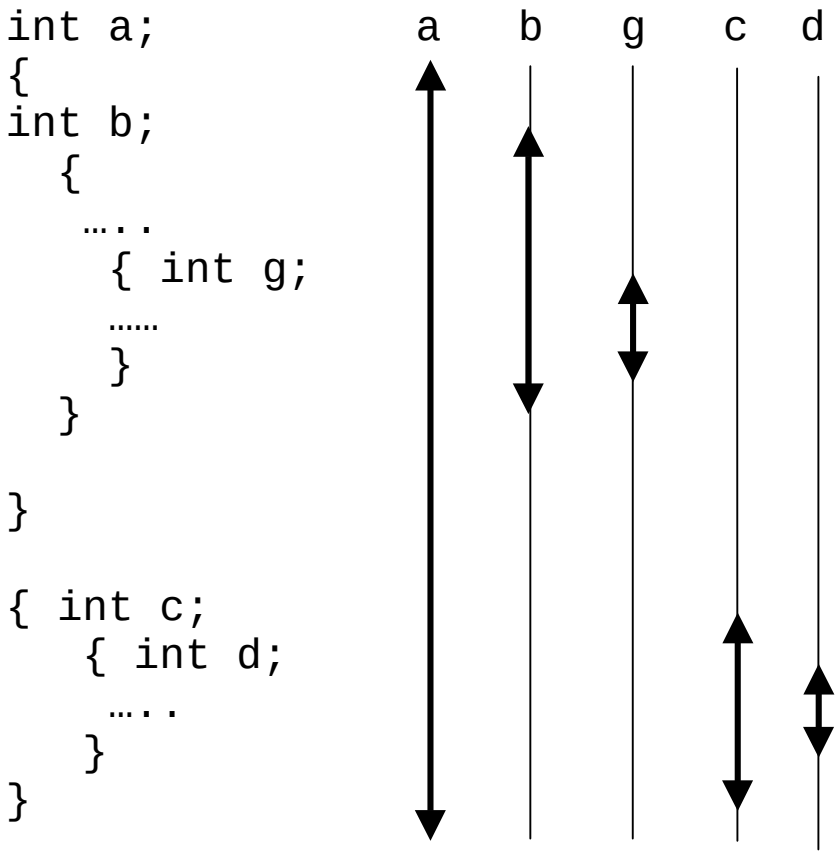
```
float a=0, val;
do{ val = f(a); if (val%2==0) printf ("val=%f, con a=%f\n", val, a);
  for (i=0;i<DR;i=i+1) { for (j=0;j<DC;j=j+1)sum = sum + am[i][j];
  } a = a + val;  } while (val<=13000);
```

E’ una buona regola di indentazione allineare parentesi graffe corrispondenti sulla stessa colonna. In questo modo anche se vi sono molte istruzioni fra una parentesi graffa aperta e la sua corrispondente chiusa risulta facile comprendere a che livello di annidamento hanno certe istruzioni rispettivamente ai costrutti di iterazione e selezioni presenti.

7.14 Il dominio di validità delle variabili, lo scope

Come e’ stato mostrato in precedenza e’ possibile tramite l’inserimento di costrutti `if`, `while`, `do-while`, `for`, etc. e anche la realizzazione di sottoprogrammi, inserire costrutti sequenza annidati. Questi definiscono dei blocchi annidati di istruzioni che possono contenere dichiarazioni di variabili distinte. Ogni blocco può avere delle variabili e queste hanno validità solo nel blocco in cui sono definite e nei blocchi sottostanti se in questi non sono definite delle variabili omonime. Il campo di validità di una variabile e' detto anche scope.

Le variabili e le costanti sono dichiarate in un certo contesto. Il contesto di validità di una variabile e' delimitato dal costrutto sequenza (coppia di parentesi graffe) nel quale sono definite. Per esempio nel grafico dell'esempio seguente la variabile a e' visibile per tutto il campo di validità mentre le altre hanno uno scope limitato.



Nel codice riportato di seguito la stessa variabile `j` e' stata definita in ogni livello successivo. Questo ha fatto si che per ogni livello di annidamento si abbia una diversa variabile `j` che va a sovrascrivere le precedenti. Questo effetto e' in alcuni rari casi desiderato ma spesso può provocare dei problemi non trascurabili. La possibilità di definire variabili e costanti in ogni punto del codice del C/C++ provoca spesso situazioni del genere.

7.15 Un esempio di utilizzo delle variabili globali e locali

Di seguito e' riportato un esempio dell'utilizzo delle variabili globali e locali a vari livelli.

```
#include "stdio.h"
#include "stdlib.h"
…….
int j=11;
……
void main(void )
{
int i;
……..
printf("globale: j=%d \n", j);
int j=7;
```

```
……….
printf("main: j=%d \n", j);
for (i=0;i<4;i=i+1)
  { int j=3;
  printf ("for: j=%d, i=%d \n", j, i);
  if (i==2)
    { int j=4;
    printf("if i==2: j=%d \n", j);
    }
  }
printf("main: j=%d \n", j);
}
```

Il risultato prodotto dall'esecuzione di tale programma e' molto significativo. Ad ogni livello viene considerata una diversa variabile `j`. Quando si hanno programmi molto lunghi, e' molto difficile sapere se vi sono state delle dichiarazioni di variabili intermedie fra l'inizio del programma e il punto nel quale si vuole utilizzare una data variabile. In tali casi e' molto facile considerare variabili omonime come la stessa variabile anche se in effetti non lo sono. Per evitare tale problema e' buona regola non utilizzare la capacita' del C/C++ di poter definire delle variabili in ogni costrutto sequenza ma di limitarsi a dichiarare le variabili nelle prime righe del programma principale e dei sottoprogrammi.

```
globale: j=11
main: j=7
for: j=3, i=0
for: j=3, i=1
for: j=3, i=2
if i==2: j=4
for: j=3, i=3
main: j=7
```

Si noti inoltre che se un programma e' implementato realizzando svariati sottoprogrammi la dichiarazione di una variabile fuori dal programma principale prima della realizzazione dei sottoprogrammi porta a rendere direttamente visibile tale variabile nei sottoprogrammi. Questo tipo di variabili vengono dette globali. La definizione di variabili globali deve essere evitata perché può generare effetti indesiderati e quindi errori di difficile identificazione, come sarà mostrato in seguito.

8 Modulo: I puntatori e vettori

I puntatori sono delle variabili che mantengono il valore dell'indirizzo di altre variabili o di altri puntatori. Le dimensioni di una variabile puntatore e quindi la sua rappresentazione in memoria dipendono dal sistema operativo, dal microprocessore, dal linguaggio e dal compilatore utilizzati.

Vi sono linguaggi che sono privi di puntatori, mentre altri, come il C/C++, dove i puntatori sono di fondamentale importanza. Il concetto di puntatore e' uno dei concetti più importanti della programmazione con i linguaggi evoluti. I linguaggi che non hanno i puntatori sono tipicamente meno flessibili rispetto a quelli che permetto il loro utilizzo. In alcuni casi il concetto di puntatore e' sostituito con alcuni surrogati.

8.1 I puntatori e operatori relativi: &, *

I puntatori sono tipizzati, questo permette di identificare il tipo di variabile ai quali puntano. La loro rappresentazione in memoria non dipende dal tipo di variabile alla quale si riferiscono. La tipizzazione dei puntatori e' utile quando si effettuano dei calcoli con il loro valore.
In C/C++, vi la possibilità di dichiarare dei puntatori neutri, cioè non tipizzati.

La dichiarazione di puntatori si effettua semplicemente anteponendo un asterisco al nome della variabile in fase di dichiarazione:

```
int *a, b=5; //dichiarazione del puntatore a e la variabile b
```

Le due variabili `a` e `b` non sono dello stesso tipo. Una e' un puntatore l'altra e' una variabile intera. Pertanto non e' possibile assegnare il valore di `b` ad `a` e viceversa.

Dopo la dichiarazione di un puntatore il suo valore non ha molto senso perché non indica/referenzia una specifica variabile. Pertanto i puntatori non devono essere utilizzati senza aver inizializzato in modo opportuno il loro valore. Per convezione il loro valore iniziale dovrebbe essere pari a `NULL`. `NULL` e' una costante simbolica pari a 0.

Data una variabile e' possibile risalire al suo indirizzo tramite l'operatore unario &. Per esempio se si ha la variabile `b`, con `&b` si intende l'indirizzo della cella di memoria utilizzata dalla variabile `a`. Pertanto in accordo alle dichiarazione dell'esempio precedente e' possibile effettuare la seguente assegnazione:

```
a = &b;
```

Dopo questa assegnazione il puntatore a punta alla cella b. Questo e' rappresentato con una freccia. Se un puntatore punta ad una cella può essere utilizzato per manipolare il contenuto di tale cella. Per esempio con *a si fa riferimento alla cella puntata dal puntatore a.

Tale notazione può essere utilizzata in espressioni:

```
int *a, b=5, c=2;
a = &b;
c = *a * c;
printf("%d, %d, %d \n", *a, b, c);
```

Si veda ad esempio la seconda assegnazione, con la quale il valore della cella puntata dal puntatore a (cioè la cella della variabile b) viene moltiplicata per il valore della cella c.
Pertanto il programma stampa a video i seguenti numeri: 5, 5, 25;

8.2 *Array* di puntatori

Si possono definire ovviamente anche *array* di puntatori sia monodimensionali che multidimensionali.

Anche in questo caso, nelle espressioni i vari operatori relativi ai puntatori (* e &) vengono applicati in base a delle regole specifiche di precedenza. Tutti gli operatori con le relative precedenze sono riportati in sessione 2.17.

Quando non e' possibile specificare esattamente il tipo di puntatore si può utilizzare il tipo void, per esempio con void* (che si legge: void pointer), che rappresenta un tipo generico di puntatore. Al momento dell'uso, questo può essere convertito nel tipo corretto per mezzo di una operazione di *casting* (conversione di tipo) esplicito.

Si possono definire dei vettori di puntatori semplicemente con:

```
int * p[10]; // un vettore di 10 puntatori ad interi
```

Questo non deve essere confuso con:

```
int (*p)[10];
```

Che specifica un puntatore p a un vettore di 10 interi.

8.3 Puntatori e stringhe

Gli operatori * e & per la gestione di puntatori sono molti utili per la gestione di vettori e quindi anche di stringhe che sono vettori di caratteri.

Le seguenti dichiarazioni di stringhe sono equivalenti:

```
1) char *buf = "baldo";
2) char buf[]="baldo";
```

In tutti e due questi casi la stringa `buf` viene realizzata come un vettore di 6 caratteri (anche nel primo caso). Sia nel primo che nel secondo caso si può fare riferimento ai singoli caratteri direttamente per mezzo dell'indice; per esempio, con `buf[3]` si identifica la cella che contiene il carattere `'d'`, il quarto carattere.

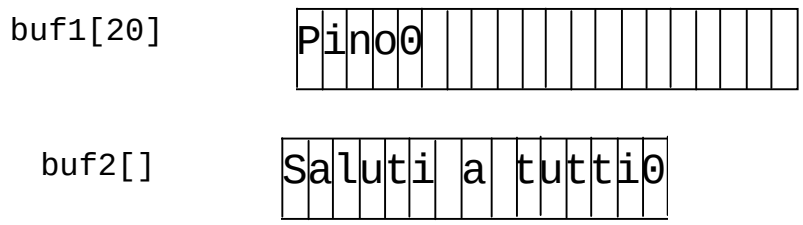
In entrambe i casi il puntatore al primo carattere, quello di indice zero può essere indicato con: `&buf[0],` oppure semplicemente con `buf.` Questa seconda notazione e' tipicamente utilizzata in C ma non in C++, dove `buf` può anche rappresentare l'intero oggetto stringa `buf` e non solo il suo puntatore.

8.4 Copia di stringhe: funzione `strcpy()`

Tipicamente le stringhe vengono manipolate in svariati modi. L'operazione più semplice e' la copia di una stringa in un'altra. Tale operazione non può essere eseguita semplicemente utilizzando l'operatore di assegnazione ed l'identificativo della stringa poiché tale identificativo rappresenta un puntatore alla stringa. Pertanto il seguente programma produce il risultato sperato:

```
char buf1[20]= "pino";

char buf2[] = "Saluti a tutti";

buf1=buf2;
```

Ma crea in effetti un problema. Prima dell'assegnazione si avevano in memoria due stringhe allocate in due are di memoria distinte. La prima conteneva la parola "pino", la seconda la frase "Saluti a tutti", come rappresentato dalla seguente figura.



Dopo l'assegnazione il puntatore `buf1`che identificava la stringa `buf1` punta all'area di memoria della stringa `buf2.` Pertanto l'area di memoria relativa alla stringa `buf1` e' perduta per sempre e non può essere più utilizzata. Le due stringhe risultano coincidenti, buf1 che prima era di 20 caratteri ora si riferisce ad un'area di memoria di 15 caratteri.

Le stringhe come i vettori si possono copiare effettuando la copia carattere per carattere oppure tramite funzioni di libreria. Come primo passo vediamo un programma che effettua la copia della stringa `buf2` nella `buf1`:

```
char buf1[20]= "pino";
char buf2[] = "Saluti a tutti";
```

.....

```
int i;
for (i=0;i<=strlen(buf2); i=i+1) buf1[i]=buf[2];
```

La funzione di libreria per effettuare la copia di stringhe deve essere utilizzata come segue:

```
strcpy(buf1, buf2); // copia di buf2 su buf1
```

8.5 Concatenazione di stringhe: funzione `strcat()`

La concatenazione di stringhe e' quell'operazione che permette di congiungere delle stringhe aggiungendo i caratteri di una dopo quelli dell'altra. Si veda per esempio il seguente spezzone di codice:

```
char buf1[20]= "pino";
char buf2[] = "Saluti a tutti";
.....
strcat(buf2, ", ");
strcat(buf2, buf1);
printf ("%s \n", buf2);
```

Che produce a stampa: `Saluti a tutti, pino`

In alcuni linguaggi, le stringhe possono essere concatenate tramite l'operatore di somma. In C/C++, e' possibile concatenare stringhe in modo semplice quando queste sono delle costanti. Le seguenti assegnazioni sono equivalenti:

```
char bufx[] = "Saluti a tutti" ", " "Gino";

char bufy[] = "Saluti a tutti, Gino";
```

8.6 Confronto di stringhe: funzione `strcmp()`

Le stringhe possono essere confrontate per verificare la loro uguaglianza o per vedere se in base all'ordine alfabetico una stringa e' maggiore (si trova dopo nel dizionario) rispetto ad un'altra. Per effettuare tale operazione di confronto si può utilizzare una funzione di libreria come riportato nel seguente esempio:

```
char buf1[20]= "pino";
char buf2[] = "Saluti a tutti";
int ret;

ret=strcmp(buf1, buf2);

if (ret>0)
   printf ("buf1 > buf2\n");
else if (ret<0)
```

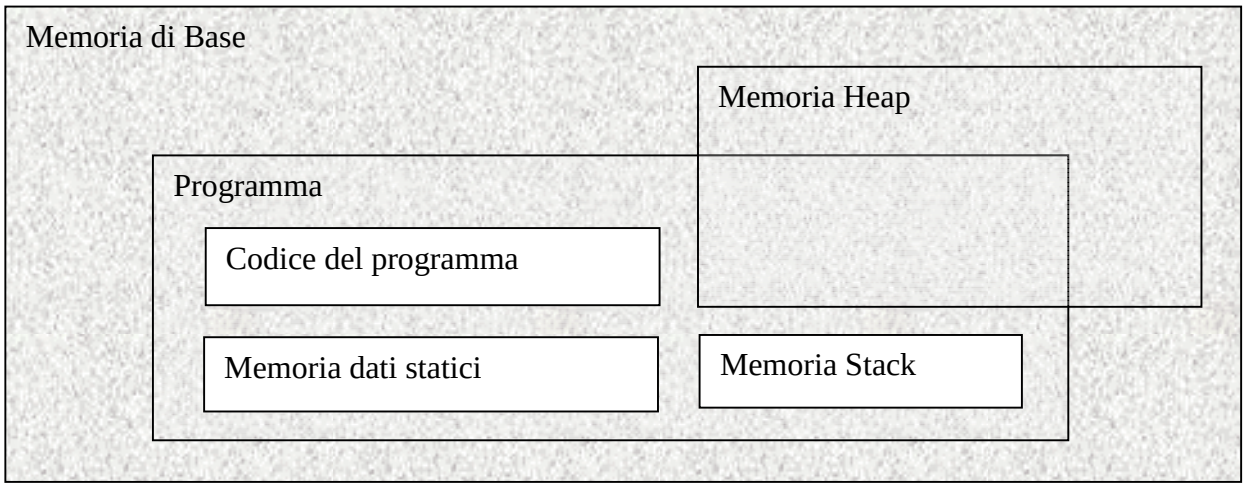
```
   printf ("buf1 < buf2 \n");
else
   printf ("buf1 = buf2 \n");
```

Nell'esempio stesso sono riportati i vari casi che permettono di capire quale delle due stringhe e' la più "leggera", cioè la prima in ordine alfabetico.

8.7 Allocazione dinamica di variabili

I puntatori sono alla base della gestione dinamica della memoria. Si parla di aspetti dinamici quando ci si riferisce a quanto può essere effettuato durante l'esecuzione del programma. Tipicamente le dichiarazioni che sono state utilizzate in precedenza danno luogo a variabili la cui presenza e' decisa direttamente durante la redazione del programma. E' inoltre possibile richiedere ed utilizzare delle variabili anche durante l'esecuzione del programma. Queste variabili possono essere utilizzate per mezzo di puntatori.

Il meccanismo con il quale si riserva una parte della memoria ad una nuova variabile direttamente durante l'esecuzione del programma viene detto di *allocazione dinamica della memoria*. Nella memoria di base del calcolatore e' riservata un'area per l'allocazione di nuove variabili. Tale area e' detta memoria *Heap* (mucchio). Tutti i programmi del calcolatore possono accedere a tale memoria.

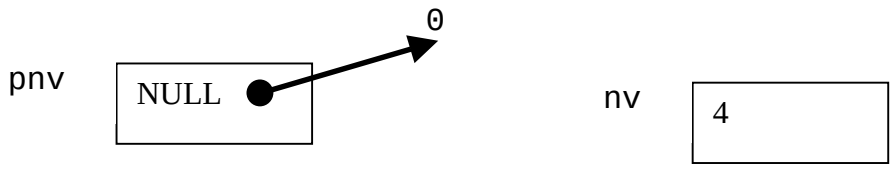


8.8 Esempio di allocazione dinamica di variabili

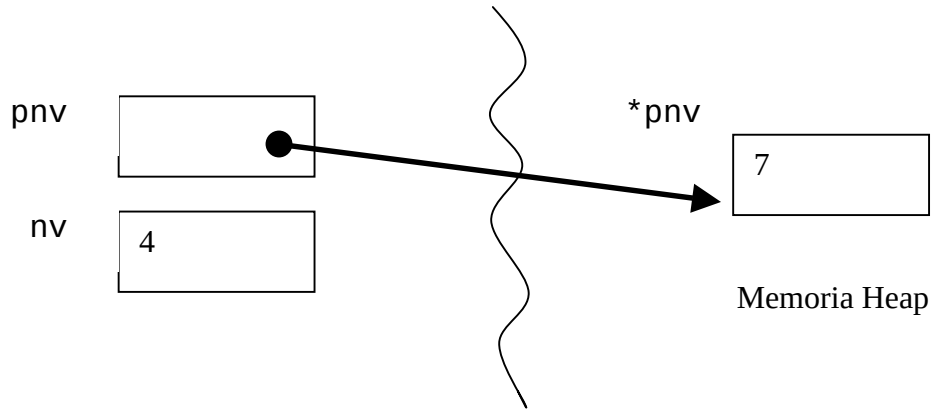
Nell'esempio che segue viene allocata la variabile intera `nv` e connessa al puntatore `pnv`. Al momento della dichiarazione di un puntatore il suo valore dovrebbe essere assegnato a NULL per convezione (cioè nullo, a niente) direttamente dal compilatore. In alcuni casi questo non viene effettuato pertanto si consiglia di effettuare tale assegnazione ad ogni dichiarazione. Si veda l'esempio che segue:

```
int nv=4, *pnv=NULL;
```

```
pnv = new int;
*pnv = nv + 3;
printf ("%d %d \n", *pnv, nv);
```



Il risultato stampato da questo esempio e': `7, 4`; la situazione descritta per via grafica sopra rappresenta lo stato prima dell'esecuzione dell'istruzione `new` (di allocazione), mentre la seguente figura descrive la situazione dopo l'allocazione e la seconda assegnazione. Sulla destra e' riportata la memoria *Heap*.



8.9 Vantaggi dell'allocazione dinamica di variabili e deallocazione

L'allocazione dinamica di variabili permette di rimandare la definizione della quantità di memoria utilizzata al momento dell'esecuzione. Questo permette di sfruttare solo la memoria strettamente necessaria. Per esempio, se si deve realizzare un programma e non si conosce a priori la quantità di elementi di un vettore che saranno utilizzati e' possibile definire un puntatore a vettore e deciderne il numero di elementi solo durante l'esecuzione, utilizzando in questo modo solo la memoria strettamente necessaria a contenere gli elementi disponibili. La soluzione statica porta a massimizzare le dimensioni del vettore e quindi a dichiarare staticamente un vettore con il numero massimo di elementi.

Un secondo vantaggio dell'allocazione dinamica consiste nel fatto che la memoria che viene allocata (riservata per certe variabili) può essere rilasciata per essere riutilizzata. Tale procedimento viene detto di deallocazione.

Si ricorda che il procedimento di allocazione assegna l'indirizzo della memoria allocata al puntatore per il quale la memoria e' stata allocata. Il procedimento inverso di deallocazione, viene eseguito con l'istruzione `delete` del C++ e provoca l'assegnazione a `NULL` del puntatore di riferimento a tale memoria.

Per esempio:

```
int nv=4, *pnv=NULL;
.........
pnv = new int;   // allocazione di un intero
.........
delete(pnv); // deallocazione, pnv ritorna NULL
```

In questo esempio e' stato dichiarato un puntatore a vettore `pnv`, ed e' stato inizializzato a `NULL`. Con l'istruzione `new` e' stata allocata una variabile intera e collegata a `pnv` assegnandogli il puntatore di tale variabile (questa e' la funzionalità della funzione `new`).
Con la chiamata alla funzione `delete()` il puntatore viene reimpostato a `NULL` e la variabile da lui puntata nella memoria *Heap* viene eliminata rilasciando la memoria che occupava per successivi riutilizzi di altre allocazioni.

Il procedimento di deallocazione viene detto anche di cancellazione. In effetti con tale meccanismo si effettua la cancellazione dell'oggetto istanziato durante l'allocazione.

8.10 L'aritmetica dei puntatori, vettori e puntatori

In C/C++, i puntatori possono essere manipolati in vario modo. Una delle operazioni più utili e' senza dubbio quella di incremento di un'unita'. Per i puntatori questa operazione ha un significato particolare. Si faccia riferimento al seguente esempio in cui si ha un vettore di `float` gestito con un puntatore all'inizio del vettore:

```
float vet[] = { 2.1, 4.3, 8.3, 10.0, 12.1, 14.0, 16.4, 18.6 };
float *ivet = vet;
```

Il puntatore `ivet` e' stato inizializzato al primo elemento del vettore `vet[],` a questo fine si poteva utilizzare anche semplicemente `vet` invece che `ivet.` Per stampare i primi cinque elementi (dallo 0 al 4 compresi) del vettore si può utilizzare il seguente segmento di codice:

```
for (i=0;i<5;i=i+1) printf ("%f, ", ivet+i);
```

Si noti come sia possibile passare a indicare l'elemento successivo nel vettore di `float` semplicemente incrementando il puntatore di una unita' invece che di 4 byte come si dovrebbe fare considerando l'effettivo ingombro in memoria delle variabili `float`.
Questo tipo di comportamento per i puntatori si può avere poiché questi sono tipizzati, cioè sono definiti in base ad un certo tipo di dato. E' sulla base di tale definizione che viene effettuato l'incremento effettivo in base a quello unitario.

Un ulteriore esempio di questa capacita' risulta evidente dal seguente codice:

```
int v1[10], v2[10], i, *vx;
```

```
i= &v1[5]-&v1[3];   // a i viene assegnato il valore 2
i= &v1[5]-&v2[3];   // i non è definito, un valore che non ha senso

vx= v2+2; // vx punta a v2[2]
```

Questi esempi chiariscono meglio la potenza dell'aritmetica dei puntatori in base al tipo.

8.11 I Riferimenti, Reference del C++

In C++, e' possibile definire delle variabili che condividono la stessa area di memoria con altre variabili, cioè sono sostanzialmente dei sinonimi di altre variabili. Nelle sessioni precedenti e' stato mostrato come sia possibile modificare il valore di una variabile attraverso un puntatore a tale variabile. Anche i puntatori sono degli strumenti che possono permettere di condividere più variabili per mezzo di punti di accesso diversi: puntatori più il nome stesso della variabile.
Nell'esempio seguente la variabile `i` viene referenziata da un'altra variabile `r` che di fatto risulta essere una sorta di *alias* della prima:

```
int i=34;
int & r= i;
```

Le variabili dichiarate come *reference* (per mezzo dell'operatore `&` fra il tipo ed il nome della variabile) possono solo essere inizializzate e quindi associate ad altre variabili ma il loro legame (valore) non può essere modificato dinamicamente (durante l'esecuzione del programma).

Per tale utilizzo si consiglia di impiegare dei puntatori.

9 Modulo: Acquisizione dati

Una delle caratteristiche fondamentali dei calcolatori elettronici e' quella di acquisire e memorizzare dati, effettuare calcoli e produrre dei risultati. A tale scopo e' di fondamentale importanza l'utilizzo di funzioni per acquisire dati dall'esterno. Nei moderni calcolatori i dati possono essere acquisiti tramite tastiera o passati tramite file. In genere nei linguaggi di programmazione vi sono delle istruzioni specifiche per l'acquisizione di dati. In C, questa operazione viene assolta da funzioni di libreria.

9.1 Acquisizione dati da tastiera, la funzione `scanf()`

Per l'acquisizione di dati da tastiera e' necessario conoscere il tipo di dato che si intende ricevere da tastiera. A questo fine e' consigliabile prima di invocare l'istruzione `scanf()` per acquisire il dato dalla tastiera, proporre un'apposita stringa sul video in modo da guidare l'utente durante l'immissione. Con il seguente esempio si richiede all'utente l'immissione di un numero intero:

```
int alfa;
……
printf("Inserire il valore di alfa: ");
scanf("%d", &alfa);
```

L'esecuzione di questo programma produce a stampa il seguente messaggio:

```
Inserire il valore di alfa: _
```

Presentando il cursore lampeggiante alla fine della linea prodotta dalla funzione `printf().` A tal punto e' possibile inserire un dato di tipo intero, che sarà recepito dalla funzione `scanf()` e assegnato da questa alla variabile `alfa`. Questo e' possibile poiché alla funzione `scanf()` le variabili vengono riferite dando il loro puntatore, per esempio `&alfa`.

In C, la procedura per ricevere dei dati opera una vera e propria conversione di formato che viene applicata quando viene letto il valore di una variabile per mezzo di funzioni tipo `scanf(), fscanf(),` secondo la seguente Tabella. Ogni descrittore di formato di ingresso può essere costituito da tre elementi nella forma `%qT`, dove tali componenti hanno il seguente significato.

- il simbolo di percentuale `%` identifica il convertitore di formato nella maschera di ingresso. In questo caso viene chiamata maschera poiché, se nella stringa sono specificati dei caratteri, il dato ricevuto in ingresso deve soddisfare il prototipo dato e, quindi, la maschera;
- `q,` se presente, assume il valore `h` per ricevere correttamente gli `short*, l` per ricevere correttamente `long*` oppure, se il tipo e' `e, f, g` per ricevere correttamente i `double*` invece che i `float*`, e `L` per ricevere correttamente i `long double*;`
- `T` indica il tipo come riportato in Tabella.

Per esempio, per leggere correttamente una variabile di tipo `unsigned long` si deve utilizzare `%lu`.

9.2 Tabella per la funzione scanf()

Nella seguente tabella sono riportati i codici per la conversione dei dati in ingresso.

T	argomento	Dato letto come
d	int*	Decimale con segno
i	int*	Decimale con segno, oppure ottale con o senza O, oppure esadecimale con o senza 0x o 0X
o	int*	Ottale, con o senza O
x	int*	Esadecimale (con o senza 0x o 0X)
u	unsigned int*	Decimale senza segno
c	char*	Carattere (senza porre \0)
s	char*	Stringa di caratteri, viene aggiunto \0
E,f,g	float*	Numero reale nelle varie forme (scientifica, float, normale)
p	void*	Puntatore far
n	int*	Puntatore near (senza segmento)
%		Carattere %

9.3 Acquisizione di una stringa da tastiera

Con il seguente esempio si richiede all'utente l'immissione di una stringa di caratteri:

```
char buf [20];
……
printf("Inserire la stringa: ");
scanf("%s", buf);
```

L'esecuzione di questo programma produce a stampa il seguente messaggio:

```
Inserire la stringa: _
```

Presentando il cursore lampeggiante alla fine della linea prodotta dalla funzione `printf()`. A tal punto e' possibile inserire la stringa che sarà recepita dalla funzione `scanf()` e assegnata da questa nel vettore di caratteri `buf[]`. Questo e' possibile poiché alla funzione `scanf()` le variabili vengono riferite dando il loro puntatore. Si ricordi infatti che scrivere `buf` equivale a scrivere `&buf[0]`. In C++, si consiglia di utilizzare la seconda notazione poiché e' piu' esplicita.

9.4 Acquisizione di caratteri da tastiera

L'acquisizione di informazioni da tastiera può essere effettuata tramite la funzione di libreria `scanf()`. Questa funzione permette di inserire dati e li rende disponibili all'interno del programma dopo che l'utente ha premuto il tasto di invio (*return* o *enter*). Spesso vi e' la necessita' di acquisire singoli caratteri da tastiera senza per questo costringere l'utente a premere il tasto di invio ad ogni inserimento. Questa modalità permette la realizzazione di menu e di scelte rapide. Questa operazione può essere effettuata tramite la funzione di libreria `getch()`, che ha come prototipo:

```
char getch(void );
```

Nell'esempio seguente tale funzione e' utilizzata per realizzare un piccolo menu con varie scelte:

```
main()
{
char car;

printf("a) scelta a  \n");
printf("b) scelta b \n");
printf("c) scelta c \n");
…….
Printf("Premi un carattere per scegliere \n");
car=getch(); // acquisizione di un carattere da tastiera


switch(car)
  { case 'a':
        printf ("Scelta a effettuata \n");
        …….. // istruzioni associate alla scelta
        break;
  case 'b':
        printf ("Scelta c effettuata \n");
        …….. // istruzioni associate alla scelta
        break;
  case 'c':
        printf ("Scelta c effettuata \n");
        …….. // istruzioni associate alla scelta
        break;
  default:
        printf ("scelta non consentita, carattere %c non valido \n",
car);
        break;
  }

/* fine del programma pincipale */
}
```

9.5 Cattura della pressione di un tasto

Se un programma deve eseguire delle operazioni e contemporaneamente verificare se l'utente desidera interromperle oppure effettuare altre cose, e' possibile verificare se e' stato premuto un tasto della tastiera senza per questo averlo richiesto in modo esplicito come con le funzioni `scanf`() o `getch`().

Il seguente programma effettua la stampa a video dei numeri interi, ogni volta che l'utente preme un tasto il programma interrompe l'esecuzione del ciclo iterativo principale per eseguire alcune istruzioni: stampare il carattere corrispondente al tasto che e' stato premuto e attendere la pressione di un secondo tasto per continuare la corsa. Con la pressione del tasto 'e' viene eseguita una istruzione break per uscire dal ciclo e quindi dal programma.

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"

void main(void )
{
int i=0; char c;

while (1)
  { printf("(%d),",i); i++;  // stampa del numero ed incremento

  if (kbhit()!=0) // verifica della pressione del tasto
    { c=getch(); // lettura del carattere dal buffer della tastiera
    printf("premuto %c\n", c); // stampa del tasto premuto

     if (c=='e') break;
    printf("Premi un tasto per riavviare");
    getch(); // attesa della pressione per il riavvio
    printf("\n");  // ritorno a capo
    }
  }
}
```

Lo studente provi il programma premendo tasti a caso sulla tastiera. Si cerchi di capire cosa accade quando sono premuti i tasti funzione.

9.6 La codifica della tastiera

Con il seguente programma e' possibile stampare a schermo la codifica ASCII di ogni tasto della tastiera. Si noti il comportamento di tutti gli altri tasti in combinazione con i tasti SHIFT (shift), CTRL (control), e ALT (Alternate). In alcuni casi la semplice pressione di un tasto provoca l'invio da parte della tastiera di piu' caratteri, si veda per esempio i tasti funzione, le frecce, etc.

```
#include "stdio.h"
```

```
#include "stdlib.h"
#include "conio.h"

void main(void )
{
int i=0; char c;

while (1)
  { c=getch();
  printf("carattere %c, decimale %d, esadecimale %x \n", c, c, c);
  }
}
```

La seguente tabella di codifica dei tasti non deve essere confusa con la tabella ASCII.

Tasto	normale		shift		Ctrl		Alt	
F1	0	59	0	84	0	94	0	104
F2	0	60	0	85	0	95	0	105
F3	0	61	0	86	0	96	0	106
F4	0	62	0	87	0	97	0	107
F5	0	63	0	88	0	98	0	108
F6	0	64	0	89	0	99	0	109
F7	0	65	0	90	0	100	0	110
F8	0	66	0	91	0	101	0	111
F9	0	67	0	92	0	102	0	112
F10	0	58	0	93	0	103	0	113
F11	0	123	0	121	0	119	0	117
F12	0	122	0	120	0	118	0	116
←	0	75	-	52	0	115	-	-
→	0	77	-	54	0	116	-	-
↑	0	72	-	56	-	-	-	-
↓	0	80	-	50	-	-	-	-
HOME	0	71	-	55	0	119	-	-
END	0	79	-	49	0	117	-	-
PGUP	0	73	-	57	0	132	-	-
PGDN	0	81	-	51	0	118	-	-
INS	0	82	-	48	0	-	-	-
DELETE	0	83	-	46	0	255	-	-
ESC	-	0	-	0	-	0	-	-
BACKSPACE	-	8	-	8	-	0	-	-
TAB	-	9	0	15	-	-	-	-
RETURN	-	13	-	13	-	10	-	-
a	-	97	-	65	-	1	0	30
b	-	98	-	66	-	2	0	48

c	-	99	-	67	-	3	0	46
e	-	100	-	68	-	4	0	32
e	-	101	-	69	-	5	0	18
f	-	102	-	70	-	6	0	33
g	-	103	-	71	-	7	0	34
h	-	104	-	72	-	8	0	35
i	-	105	-	73	-	9	0	23
j	-	106	-	74	-	10	0	36
k	-	107	-	75	-	11	0	37
l	-	108	-	76	-	12	0	38
m	-	109	-	77	-	13	0	50
n	-	110	-	78	-	14	0	49
o	-	111	-	79	-	15	0	24
p	-	112	-	80	-	16	0	25
q	-	113	-	81	-	17	0	16
r	-	114	-	82	-	18	0	19
s	-	115	-	83	-	19	0	31
t	-	116	-	84	-	20	0	20
u	-	117	-	85	-	21	0	22
v	-	118	-	86	-	22	0	47
w	-	119	-	87	-	23	0	17
x	-	120	-	88	-	24	0	45
y	-	121	-	89	-	25	0	21
z	-	122	-	90	-	26	0	44
[	-	91	-	123	-	0	-	-
\	-	92	-	124	-	28	-	-
]	-	93	-	125	-	29	-	-
'	-	96	-	126	-	-	-	-
0	-	48	-	41	-	-	0	129
1	-	49	-	33	-	-	0	120
2	-	50	-	64	0	3	0	121
3	-	51	-	35	-	-	0	122
4	-	52	-	36	-	-	0	123
5	-	53	-	37	-	-	0	124
6	-	54	-	94	-	30	0	125
7	-	55	-	38	-	-	0	126
8	-	56	-	42	-	-	0	127
9	-	57	-	40	-	-	0	128
*	-	42	-	-	0	114	-	-
+	-	43	-	43	-	-	-	-
-	-	45	-	95	-	31	0	130
=	-	61	-	43	-	-	0	131
@	-	44	-	60	-	-	-	-
/	-	47	-	63	-	-	-	-
;	-	59	-	58	-	-	-	-
~	-	96	-	126	-	-	-	-

9.7 Caratteri di controllo

La seguente tabella illustra la codifica dei caratteri di controllo, ottenuti per combinazione del tasto `Ctrl` (control) e di altri caratteri della tastiera.

Ctrl	Dec	Hex	Car	Code	Descrizione
^@	0	0		NUL	NULL, nil, nullo
^A	1	1		SOH	partenza blocco
^B	2	2		STX	start trasmission
^C	3	3		ETX	control break, fine testo
^D	4	4		EOT	end of transmission
^E	5	5		ENQ	interrogazione
^F	6	6		ACK	acknowledge
^G	7	7	\a	BEL	beep
^H	8	8	\b	BS	backspace
^I	9	9	\t	HT	tabulazione orizzontale
^J	10	A	\n	LF	newline (cambio linea)
^K	11	B	\v	VT	tabulazione verticale
^L	12	C	\f	FF	formfeed, alim. carta
^M	13	D	\r	CR	return, carriage return
^N	14	E		SO	codice semplice
^O	15	F		SI	codice normale
^P	16	10		DLE	uscita trasmissione
^Q	17	11		DC1	controllo periferica 1
^R	18	12		DC2	controllo periferica 2
^S	19	13		DC3	controllo periferica 3
^T	20	14		DC4	controllo periferica 4
^U	21	15		NAK	non ACK
^V	22	16		SYN	sincronismo
^W	23	17		ETB	fine del blocco
^X	24	18		CAN	annullamento
^Y	25	19		EM	fine del supporto
^Z	26	1A		SUB	EOF, end of file
^[	27	1B		ESC	escape
^\	28	1C		FS	separatore di file
^]	29	1D		GS	separatore di gruppo
^^	30	1E		RS	separatore di record
^_	31	1F		US	separatore di unita'

9.8 Programma principale e suoi parametri

Spesso e' utile realizzare dei programmi che siano in grado di accettare dei parametri di esecuzione direttamente da linea di comando. Per esempio il programma che segue e' in grado di leggere le stringhe che gli vengono passate con la linea di comando che l'utente scrive per mettere in esecuzione il programma stesso. Per esempio:

```
C:\> mioprogramma pippo pluto
```

Dove `mioprogramma` e' il nome stesso del programma realizzato dal programmatore. A questo fine, in C/C++ anche il corpo del programma principale `void main(void)` può avere dei parametri che devono seguire una convenzione:

```
/* programma mioprogramma */
.........
void main(int argc, char ** argv)
{ int i;
for (i=0;i<argc;i++) printf("Argomento %d> %s\n",i, argv[i]);
}
```

Come risultato dell'esecuzione del programma di esempio si ha:

```
Argomento 0> C:\mioprogramma.EXE
Argomento 1> pippo
Argomento 2> pluto
```

In C e' possibile restituire al sistema operativo, alla fine dell'esecuzione del programma, un numero per mezzo dell'uso delle chiamate del linguaggio `return()` o `exit()` se il `main()` non e' stato definito `void main(….)`.

10 Modulo: I sottoprogrammi

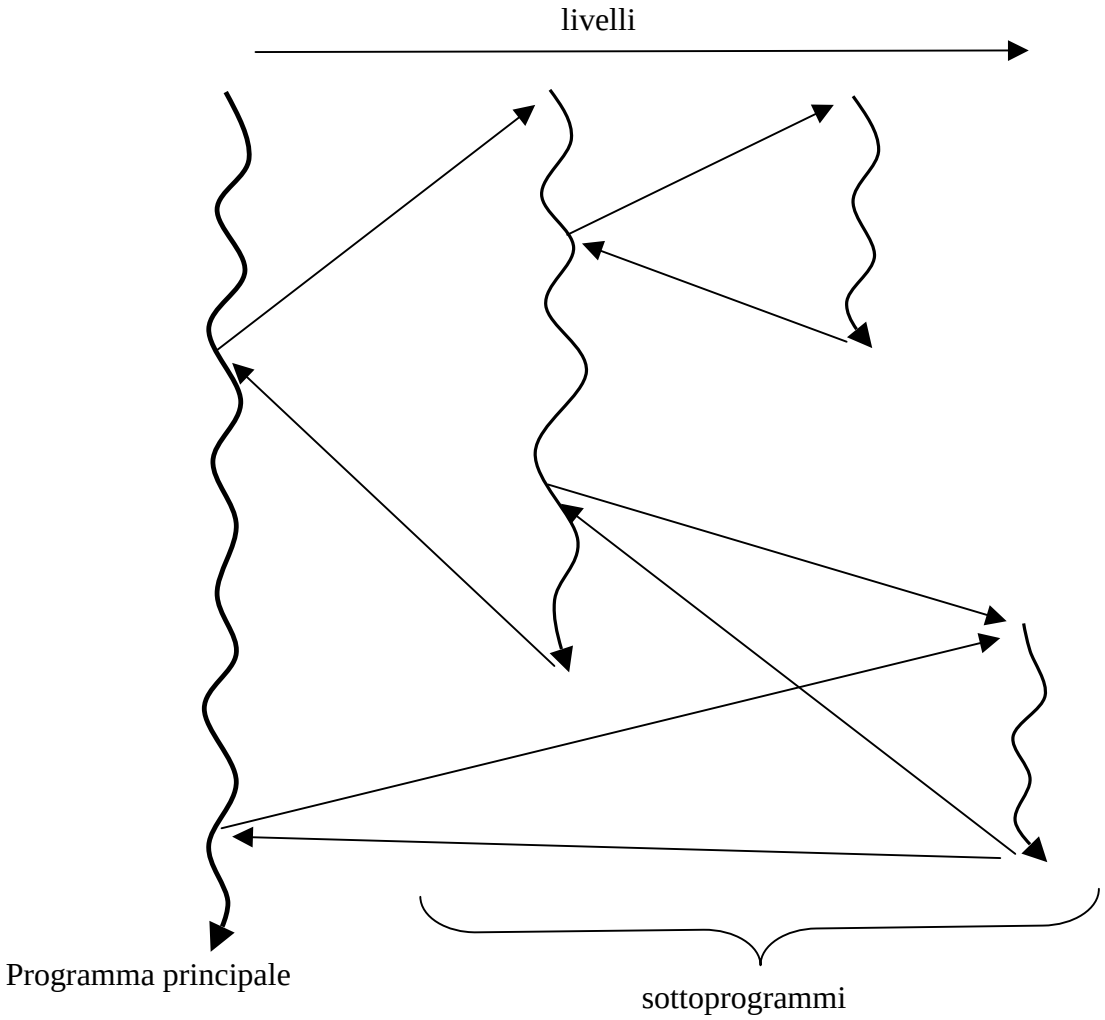
Comunemente la realizzazione di un algoritmo prevede l'utilizzo di algoritmi più semplici. Questi spesso costituiscono porzioni che possono essere riutilizzate più volte nello stesso o in altri contesti. Questo meccanismo può essere utilizzato per suddividere l'algoritmo in un insieme di problemi più semplici, che una volta risolti possono portare per composizione a risolvere il problema completo iniziale. Sorge pertanto l'esigenza di un metodo per la definizione e l'aggregazione delle istruzioni: blocchi o sequenze, procedure e funzioni. Questo meccanismo si realizza con la formalizzazione di sottoprogrammi.

La forma elementare per la suddivisione/raggruppamento di istruzioni e' il costrutto sequenza per la realizzazione di un blocco di istruzioni. Il blocco delimita un gruppo di istruzioni, ma non ha un identificativo che lo contraddistingue. In C, i delimitatori di blocco di istruzioni sono costituiti dai simboli `{ }`. A livello intuitivo, un sottoprogramma e' pertanto un blocco/sequenza di istruzioni identificato da un nome, che viene evocato al momento della chiamata. A tal punto, il programma chiamante trasferisce al sottoprogramma i dati necessari all'elaborazione secondo una convenzione posizionale. I sottoprogrammi si possono classificare in procedure e funzioni. Le differenze sono discusse nelle sessione successiva.

10.1 Sottoprogrammi: procedure e funzioni

I sottoprogrammi si dividono in procedure e funzioni. Queste sono particolari strutture di controllo che, a seguito della loro invocazione, alterano il flusso del programma.

Un programma principale può utilizzare un sottoprograma per effettuare determinate operazioni, questo a sua volta puo' utilizzare un altro sottoprogramma per effettuarne una parte, etc. I sottoprogrammi possono infine essere utilizzati da vari punti del programma sia dal programma pincipale che dai sottoprogrammi. Uno stesso sottoprogramma puo' pertanto essere utilizzato in piu' punti di un programma indipendentemente dal livello dal quale e' chiamato. A questo fine e' importante che i singoli sottoprogrammi lavorino su variabili proprie o su copie, comunque solo in modo controllato.



10.2 Struttura dei sottoprogrammi

Comunemente un sottoprogramma (procedura o funzione) C ha la seguente struttura espressa in EBNF:

```
<intestazione del sottoprogramma>
{
[<definizione di tipi locali>]
...........
[<dichiarazione di variabili locali>]
[<Istruzioni>]
}
```

Un sottoprogramma e' costituito da un'intestazione, seguita dal simbolo { che marca l'inizio del sottoprogramma. Il sottoprogramma termina con la corrispondente parentesi chiusa }. Dentro il corpo del sottoprogramma vi possono essere definizioni di nuovi tipi, quindi la dichiarazione di variabili locali e le

istruzioni stesse del sottoprogramma. Queste sono messe fra parentesi quadre poiché possono essere omesse.

Come esposto in sessione 7.14 le variabili definite localmente non sono visibili fuori dal costrutto sequenza nel quale sono definite.

10.3 Intestazione dei sottoprogrammi

L'intestazione contiene la dichiarazione del tipo restituito, che può essere omessa quando coincide col tipo `int`. In effetti vi sono due modalità di realizzare l'intestazione. La prima segue lo schema:

```
<tipo> <nome funzione>
        ( <nome argomento1>, ... <nome argomentoN> )
<tipo1> <nome argomento1>;
<tipo2> <nome argomento2>;
.....
<tipoN> <nome argomentoN>;
```

Questa forma e' ormai obsoleta. La seconda modalità per la specifica delle intestazioni dei sottoprogrammi e' registrata dallo standard ANSI 89. Nell'intestazione il nome del sottoprogramma e' seguito nella maggior parte dei casi dagli argomenti posti fra parentesi tonde:

```
<tipo> <nome funzione>(<tipo1> <nome argomento1>,
                       <tipo2> <nome argomento2>,..
                       <tipoN> <nome argomentoN>)
```

In seguito sarà utilizzata solo la seconda versione per specificare l'intestazione dei sottoprogrammi.

10.4 Sottoprogrammi: procedure e funzioni

Un sottoprogramma può essere una procedura o una funzione. Questi si differenziano per la diversa intestazione.
Le procedure sono sottoprogrammi che non hanno un tipo.

Il sottoprogramma può essere privo di parametri nel qual caso e' necessario scrivere la parola chiave del linguaggio: `void`. Questa parola chiave significa vuoto ed e' utilizzata anche in altri casi. Nella dichiarazione dei sottoprogrammi può essere utilizzata per specificare che il sottoprogramma non ha parametri oppure che non ha un tipo.

In C, il nome dato ad un sottoprogramma non può essere assegnato ad altri sottoprogrammi. In C++, questa restrizione e' stata superata permettendo la specifica di sottoprogrammi con lo stesso nome a patto che questi abbiano un diverso numero o tipo di parametri. Questo meccanismo viene detto di *overloading*. Tale meccanismo e' assolutamente indispensabile quando si vanno a realizzare delle gerarchie di classi per mezzo della definizione di relazioni di specializzazione. Si veda a questo riguardo il manuale di programmazione in C++ oppure il libro di Stroustrup citato in sessione 21.

10.5 Sottoprogrammi: Le funzioni

Le funzioni sono particolari sottoprogrammi per i quali e' richiesta l'indicazione del tipo di dato restituito in uscita, il tipo di dato che caratterizza la funzione. Le funzioni, essendo tipizzate, possono essere utilizzate come funzioni matematiche in espressioni e quindi a destra del simbolo di assegnazione o anche in condizioni di costrutti di selezione o iterazione.

A titolo di esempio viene riportata la realizzazione di una semplice funzione per effettuare la somma di due interi `int somma(int, int),` che riceve i due addendi come parametri in ingresso e restituisce il risultato della somma come numero intero direttamente sul valore di ritorno della funzione.

Si noti che per restituire il valore si utilizza la parola chiave `return` che chiude la funzione e rimanda il controllo all'istruzione successiva alla chiamata. Alla fine dell'esecuzione di tale funzione, il valore assunto dalla funzione e' pari al valore utilizzato come parametro della parola chiave `return`. Nel corpo del sottoprogramma non vi sono limitazioni al numero di chiamate alla funzione `return`().

```
int somma(int A, int B)
  {
  return(A + B);
  }
```

La porzione di codice che include la chiamata al sottoprogramma si chiama programma chiamante. Il seguente spezzone di codice riporta un esempio di programma chiamante che utilizza la funzione `somma()`:

```
int C,X,Y;
int somma(int, int);  // prototipo della funzione

.........
X = 5;
Y = 4;
C = somma(X, Y);   /* C = X + Y */
.........
```

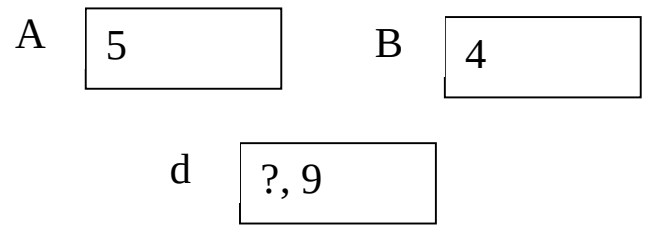
Si noti che prima dell'utilizzo della funzione o di un sottoprogramma e' necessario riportare il prototipo della funzione stessa. Il prototipo di un sottoprogramma e' una dichiarazione che specifica i tipi coinvolti nella chiamata e il loro ordine. Questa dichiarazione permette al compilatore di effettuare un controllo di consistenza fra i tipi delle variabili coinvolte nella chiamata e il sottoprogramma stesso.
Le funzioni, come tutti i sottoprogrammi possono essere privi di parametri. Per esempio la funzione `rand()` che produce un numero casuale fra 0 e il massimo della dinamica degli interi; che ha come prototipo:

```
int rand(void);
```

Al momento dell'esecuzione del programma, quando viene invocata la procedura `somma()` l'esecuzione passa a tale porzione di codice effettuando un cambio di contesto. L'operazione di cambio di contesto e' particolarmente costosa dal punto di vista computazionale poiché implica l'utilizzo di un diverso set di variabili e quindi di una diversa area di memoria. Anche il codice stesso, che deve essere eseguito spesso, non si trova in prossimità del programma chiamante pertanto si ha anche un cambio dell'IP (instruction pointer) nella CPU.

Nel nuovo contesto vengono assegnate alle variabili temporanee (che rappresentano i parametri della funzione) i valori attuali delle variabili utilizzate nella chiamata. Pertanto durante il cambio di contesto sono state allocate delle variabili, con relativo costo computazionale:

```
int somma(int A, int B)
  {
  int d;
  d = A + B;
  return d;
  }
```



Per tale ragione `A` assume il valore di `X` e `B` quello di `Y`. L'assegnamento non viene fatto in base al tipo ma solo alla posizione e quindi all'ordine con il quale le variabili nella chiamata sono organizzate rispetto alla struttura stessa dell'intestazione della funzione.
Nella versione soprastante della funzione `somma()` (del tutto equivalente alla precedente versione dal punto di vista del risultato) e' stata dichiarata una variabile temporanea `c` per il calcolo della somma. `A, B, d` sono variabili temporanee allocate in una particolare area di memoria detta *Stack*. Si veda sessioni 1.17, e 10.19.

La variabile `d` e' utilizzata per memorizzare la somma dei due numeri. Per cui prima del calcolo della somma il suo valore non e' certo ma dopo assume il valore 9, come riportato in figura.
La funzione viene conclusa con l'esecuzione dell'istruzione `return` che rimanda il controllo al programma chiamante e permette di utilizzare il valore di ritorno come il valore della funzione stessa. Pertanto nella seguente chiamata il valore di 9 viene assegnato alla variabile `C` del programma chiamante con la seguente assegnazione:

```
C = somma(X, Y);   /* C = X + Y */
```

Con il ritorno al programma chiamante la funzione viene chiusa. Con questo si intende che la funzione termina e la memoria utilizzata per le eventuali variabili temporanee che sono state allocate durante la sua esecuzione viene rilasciata. Il ritorno al contesto del programma chiamante deve essere considerato parte del costo computazionale della chiamata a sottoprogramma.

10.6 Sottoprogrammi: le procedure

Le procedure sono sottoprogrammi che possono avere dei parametri ma che non restituiscono in uscita nessun valore. Le procedure non sono tipizzate e quindi non possono essere utilizzate come funzioni matematiche in espressioni a destra del simbolo di assegnazione.

La funzione `somma()` proposta come esempio nella presentazione delle funzioni può essere realizzata anche come procedura.

```
void psomma(int A, int B, int * C)
   {
   *C = A + B;
   return;
   }
```

Si noti l'utilizzo dell'istruzione `return`. La parola chiave `return` può essere utilizzata anche nelle procedure, senza specificare il suo parametro. In tal caso provoca l'uscita della procedura per tornare al programma chiamante. Nel corpo della procedura non vi sono limitazioni al numero di volte che si può utilizzare tale istruzione per uscire dal sottoprogramma. In questo caso tale istruzione non può avere parametri. L'uso di parametri provoca un errore sintattico nel contesto di una procedura visto che questa non ha valori di ritorno. Nel caso della procedura `psomma()` la parola chiave `return` poteva anche non essere specificata visto che il sottoprogramma quando arriva all'ultima istruzione della procedura chiude il sottoprogramma per tornare al programma chiamante. Con la parola chiude si intende che il sottoprogramma termina ed che la memoria utilizzata per le eventuali variabili temporanee che sono state allocate durante la sua esecuzione viene rilasciata.

La procedura realizzata sopra deve essere utilizzata nel seguente modo:

```
int C,X,Y;
void psomma(int, int, int*); // prototipo del sottoprogramma
.........
psomma(X,Y,&C);   /* C = X + Y */
.........
```

Per realizzare sottoprogrammi non tipizzati, cioè procedure deve essere utilizzato come tipo della funzione la parola chiave `void`. Si noti che in C la distinzione fra procedure e funzioni non e' cosi' netta come in altri linguaggi, come per esempio il Pascal, che presentano parole chiave diverse per la loro definizione. Infatti, in C e' consentito chiamare le funzioni come se fossero procedure, cioè senza utilizzare il valore restituito.

Si noti che l'ultimo parametro della procedura `psomma()` e' un puntatore a intero. Per tale ragione nella chiamata a tale procedura e' stato passato l'indirizzo della variabile `C`, con `&C`.

10.7 Passaggio per valore e indirizzo

L'intestazione del sottoprogramma include sia il tipo di argomenti che devono essere passati, sia il tipo di dato restituito, se si tratta di funzione. Dopo l'intestazione, segue la dichiarazione delle variabili che vengono utilizzate all'interno del sottoprogramma e la parte procedurale. Gli argomenti che costituiscono l'interfaccia del sottoprogramma, che vengono utilizzati per progettare funzioni e procedure, sono detti parametri formali. All'interno della parte procedurale di un programma o di un suo sottoprogramma può essere presente una chiamata a sottoprogrammi. Nel corso della chiamata avviene l'assegnazione dei valori necessari all'esecuzione del sottoprogramma. Gli argomenti utilizzati dal programma chiamante, che devono risultare congruenti con i parametri formali del sottoprogramma, sono detti parametri attuali. I parametri attuali inseriti nella chiamata forniscono i valori dei parametri formali del sottoprogramma quando questo viene eseguito. Non e' necessario che i nomi degli argomenti siano gli stessi all'esterno e all'interno del sottoprogramma. Deve esserci tuttavia congruenza, nel senso che il numero di argomenti passati, l'ordine in cui essi sono elencati ed i tipi di dati devono coincidere. In alcuni casi il numero dei parametri può non coincidere se si utilizzano specifici costrutti del C++.

In generale, vi sono almeno due modi diversi per trasferire i parametri: per valore e per indirizzo. Il passaggio per valore comporta la duplicazione del valore del parametro attuale. In pratica in questo modo si garantisce che il parametro attuale non sarà modificato dal sottoprogramma. Il passaggio per indirizzo permette al sottoprogramma di lavorare direttamente sulla variabile trasmessa come parametro attuale, e pertanto le modifiche apportate all'argomento diventano visibili anche al programma chiamante. In genere si usa la prima modalità per gli argomenti in ingresso e la seconda per quelli in ingresso/uscita. Tuttavia, se la modifica dei parametri di ingresso non e' importante, o se addirittura il sottoprogramma non opera modifiche sugli ingressi, allora la trasmissione per indirizzo e' particolarmente consigliata soprattutto per strutture dati di grosse dimensioni, per risparmiare memoria e tempo di elaborazione come sarà discusso quando sarà mostrato l'uso della Stack nel passaggio dei parametri (si veda sessione 10.19).

Nel linguaggio C esiste solo la modalità di passaggio per valore. Per realizzare la trasmissione per indirizzo, il programmatore deve ricorrere all'uso dei puntatori e degli operatori unari `*` e `&`. In questo caso si parla di passaggio del valore dell'indirizzo, cioè quando e' necessario passare un argomento per indirizzo, si passa il valore del puntatore all'argomento in questione. Il sottoprogramma chiamato utilizza il puntatore per accedere direttamente alla variabile. Un esempio e' la procedura `psomma()` del paragrafo precedente. Si noti che, poiché `C` e' un puntatore ad una variabile intera, all'interno di tale procedura, per riferirsi alla variabile puntata da `C`, si deve utilizzare `*C`, che rappresenta la variabile puntata da `C`. Nella chiamata si e' utilizzato `&C`, che e' l'indirizzo di della variabile `C`.

In C++ e' stata aggiunta anche la modalità di passaggio per indirizzo tramite l'utilizzo dei riferimenti.

10.8 Passaggio per il valore dell'indirizzo

Ogni funzione restituisce un risultato del tipo dichiarato in fase di definizione e che risulta associato al suo nome. In C, una funzione può restituire un valore in uscita mediante l'istruzione `return`. Storicamente, il linguaggio C offriva l'opportunità di non specificare il tipo restituito per funzioni intere, ma questa abitudine viene sconsigliata dal recente standard ANSI. E' frequente pero' la necessita' di avere più di un parametro di ritorno, cioè di poter ricevere dal sottoprogramma chiamato più di un valore di

ritorno. In questo caso si deve utilizzare una diversa tecnica anche perché in C non e' possibile passare le variabili per indirizzo.

Per tale ragione si utilizza lo stratagemma di passare il valore dell'indirizzo come parametro. Passando il valore dell'indirizzo ad un sottoprogramma e' possibile accedere direttamente alla cella della variabile utilizzata nel programma principale.

10.9 Esempio: Passaggio per il valore dell'indirizzo

Il seguente esempio illustra le modalità per produrre due risultati in uscita da una funzione che implementa una calcolatrice:

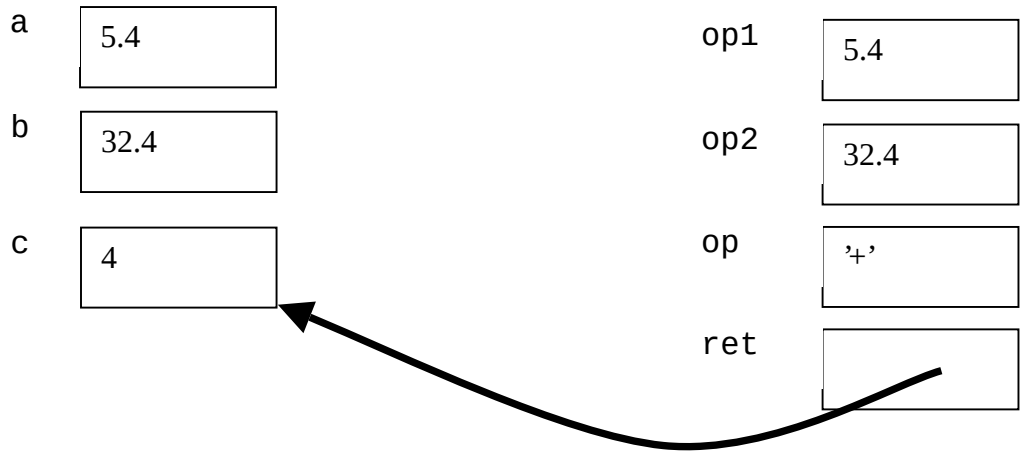
```
int calcolatrice(float op1, char op, float op2, float * ret )
{
switch (op)
   { case '+':
        *ret = op1 + op2; break;
   case '-':
        *ret = op1 - op2; break;
   case '*':
        *ret = op1 * op2; break;
   case '/':
        *ret = op1 / op2; break;
   default :
        printf("Operatore %c non ammesso!\n", op);
        return(1);
   }
return(0);
}
```

La funzione `calcolatrice()` può essere utilizzata per effettuare diverse operazioni in base al parametro `operazione` che ne specifica il tipo. Il risultato viene calcolato e posto nella cella puntata dal puntatore `ret` a `float`. Se l'operazione non e' consentita il valore della variabile puntata da `ret` rimane immutato e la funzione ritorna un valore diverso da 0.

```
int calcolatrice (float, char, float, float *);
……
int test;
float a=5.4, b=32.4, c=4;
……
test = calcolatrice (a, '+', b, &c);
if (test==0) printf ("Risultato pari a %f \n", c);
```

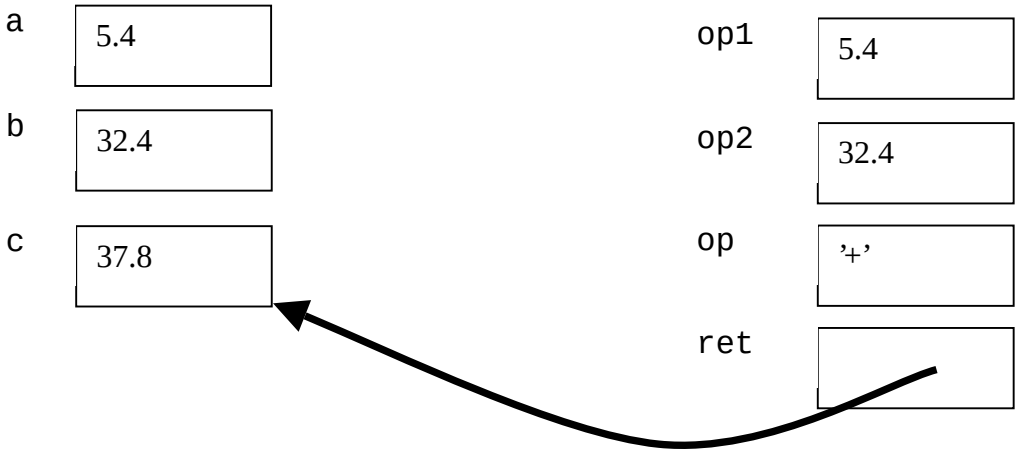
Pertanto la situazione delle celle di memoria, appena effettuata la chiamata ma prima dell'esecuzione delle istruzioni nella procedura e' la seguente.

A sinistra sono riportate le celle di memoria del programma chiamante mentre a destra quelle della



procedura allocate in forma temporanea nello *stack*. Si noti che il puntatore `ret` punta direttamente alla variabile `c` del programma chiamante. Quando si fa riferimento a `*ret` e' come utilizzare direttamente `c`.

Dopo l'esecuzione dell'operazione di somma ma prima della chiusura della procedura si ha la seguente situazione, nella quale e' chiaro che la somma ha avuto effetto diretto sulla variabile `c` del programma principale.



Infine, dopo la chiusura del sottoprogramma la memoria occupata dalle celle a destra (nello *stack*) viene liberata e ritorna ad essere disponibile per altre allocazioni temporanee. Questi passi hanno mostrato come e' possibile ottenere dei risultati da una procedura anche se questi non sono semplicemente passati indietro al programma chiamante come valore di ritorno della funzione. Questo e' stato possibile passando il valore di un puntatore: l'indirizzo di una variabile che e' stata predisposta per ricevere il risultato, la variabile `c` in questo caso.

10.10 Passaggio per Indirizzo, uso dei Reference del C++

In C non il passaggio di parametri per indirizzo e' una modalità non disponibile. Questa limitazione e' stata rimossa in C++ dove viene messa a disposizione la modalità del passaggio di parametri per indirizzo o meglio in accordo al gergo del C++, per riferimento. Con tale modalità nella procedura viene utilizzata direttamente la cella di memoria della variabile che passa per indirizzo ma tale variabile viene ad assumere un nome diverso, un alias. Nell'esempio che segue, la variabile a della procedura `stima()` viene passata per indirizzo, le variabili `b` e `c` sono passate per valore; la `b` per il valore della variabile, la `c` come valore dell'indirizzo. Tali modalità si riflettono nella chiamata stessa della procedura.

```
void stima(int & a, int b, int * c)
{
int d=5;
a = a + d ;     b = b + d;      *c = *c + d;
printf ("a=%d, b=%d, *c=%d\n", a, b, *c);
}
```

Dalla chiamata della procedura, osservando i parametri attuali, si nota come sia semplice passare una variabile per indirizzo invece che utilizzare il passaggio per valore dell'indirizzo.

```
void main(void)
{
int x=3, y=10, z=25;
stima(x, y, &z); printf ("x=%d, y=%d, z=%d\n", x, y, z);
stima(x, y, &z); printf ("x=%d, y=%d, z=%d\n", x, y, z);
}
```

Eseguendo il programma, e' chiaro come le modifiche apportate sulla variabile a della procedura (la `x` del programma chiamante) si sono ripercosse sulla variabile `x` del programma chiamante, come si ha per la variabile `z` che e' stata modificata tramite il puntatore `c` all'interno del sottoprogramma.

```
a=8, b=15, *c=30
x=8, y=10, z=30
a=13, b=15, *c=35
x=13, y=10, z=35
```

La variabile `y` e' stata passata per valore e pertanto dentro la procedura si fa riferimento solo ad una sua copia, variabile `b`, che al momento dell'uscita dal sottoprogramma viene rilasciata e non produce cambiamenti sul corrispondente parametro attuale della procedura, la variabile `y` del programma chiamante.

10.11 Dominio di validità di Variabili e Sottoprogrammi

L'uso dei sottoprogrammi obbliga a scegliere una strategia per la dichiarazione delle variabili. Le variabili dichiarate all'interno di un sottoprogramma sono variabili locali, che esistono solo finche' il sottoprogramma e' in esecuzione, non sono visibili dall'esterno del sottoprogramma. Molti linguaggi, come Pascal e C, permettono di dichiarare variabili con campo di validità non locale, anche se attraverso modi diversi. In certi casi il dominio di validità di una variabile può essere esteso a tutto il programma; in tale caso si parla di *variabili globali*.

Le *variabili globali* possono essere impiegate al posto dei parametri di un sottoprogramma, sia per acquisire dati in ingresso che per fornire i risultati. Ogni sottoprogramma può infatti essere trasformato in una procedura priva di parametri. Tuttavia, l'uso di sottoprogrammi con trasmissione dei parametri basata su variabili globali e' praticamente sconveniente, poiché rende il loro utilizzo troppo dipendente dalle dichiarazioni delle variabili e vanifica l'uso dei sottoprogrammi come veicolo di riuso del codice. Un sottoprogramma che utilizza variabili globali può essere reimpiegato in un altro contesto, solo se nel nuovo contesto sono presenti le stesse variabili globali, altrimenti si deve intervenire per modificarne il codice. Una buona norma e' dunque quella di evitare/limitare l'uso delle variabili globali, soprattutto nel caso di progetti non elementari.

In C non esiste la possibilità di nidificare i sottoprogrammi, ma solo quella di nidificare i blocchi (i costrutti sequenza), che possono avere variabili anche dichiarate localmente. Tutte le variabili dichiarate all'interno di un sottoprogramma sono locali, anche quelle dichiarate nel `main(void).` Eventualmente può variare il grado di località a seconda che la variabile sia dichiarata all'inizio del sottoprogramma o all'interno di un suo blocco. Il C ammette la dichiarazione di variabili al di fuori di sottoprogrammi. Queste variabili hanno come dominio tutte le funzioni che seguono la dichiarazione. Le variabili globali devono quindi essere dichiarate in testa al file. Mediante la parola chiave `extern` e' possibile fare riferimento a variabili dichiarate in un altro file o modulo. Queste devono essere dichiarate fuori dal `main(void)` e da ogni sottoprogramma, oppure devono avere la parola chiave `public`.

Variabili dichiarare public in un certo costrutto sequenza/blocco sono in effetti visibili anche dai livelli soprastanti. Questo tipo di dichiarazione deve essere utilizzata con estrema parsimonia, solo per uscire da situazioni complesse. In generale il suo utilizzo non e' necessario, si possono sempre fare delle modifiche in modo da risolvere il problema senza utilizzare tale soluzione.

10.12 Sottoprogrammi: Problemi in parametri di ingresso/uscita

Di seguito viene riportato un esempio in cui non e' stato adottato un uso corretto del passaggio dei parametri. Infatti nella procedura `riceve_parametri()` la variabile `j` non e' stata passata per indirizzo. I punti che permettono di correggere l'errore sono commentati.

Risultato di questa ``dimenticanza'' e' che il sottoprogramma tenta invano di modificare il valore di `j, i` per il programma chiamante:

```
Passa-parametri: prima i = 32767
```

```
Riceve-parametri: inizio j = 32767
Riceve-parametri: fine   j = 21923
Passa-parametri: dopo  i = 32767
```

I punti da correggere per ottenere la versione che utilizza correttamente il passaggio dei parametri sono svariati e sono stati commentati e colorati in giallo.

```
void riceve_parametri(unsigned /* * */j)
{
printf("Riceve-parametri: inizio j = %d\n", /* * */j);
srand((unsigned)time(NULL));

/* * */j = rand();
printf("Riceve-parametri: fine   j = %d\n", /* * */j);
}




void main(void)
{
unsigned i;

i = RAND_MAX;

printf("Passa-parametri: prima i = %d\n", i);

riceve_parametri(/* & */i);

printf("Passa-parametri: dopo  i = %d\n", i);
}
```

Con tali modifiche la variabile J viene passata per il valore dell'indirizzo.

10.13 Sottoprogrammi: occultamento di variabile Globale

In questa sessione viene mostrato uno dei possibili problemi dell'uso improprio o scorretto delle variabili globali e dei loro effetti collaterali.
Si consideri il seguente programma nel quale e' stata dichiarata una variabile globale `N`, ma all'interno della funzione `piu()` questa e' stata ridefinita.

```
int N;   /* variabile globale */
.........
int piu(int j)
  { int N;
  return N + j;
  }
.........
```

```
void main(void)
{
int i;
N = 0;

for (i = 1; i < 10; i++)
  { N = piu(i);
  printf("%d - N = %d\n", i, N);
  }
}
```

Ciò comporta che all'interno di tale funzione viene modificata solo la variabile dichiarata localmente, come mostrato dalla seguente traccia di esecuzione:

```
1 - N = 26
2 - N = 27
3 - N = 28
........
```

Dove ``26" e' stato prodotto a causa del valore iniziale della cella di memoria utilizzata per contenere `N` nello *Stack*. Pertanto tale valore e' del tutto casuale.

10.14 Sottoprogrammi: assegnazione a variabile globale

In questa sessione viene mostrato un altro dei possibili problemi dell'uso improprio o scorretto delle variabili globali e dei loro effetti collaterali. Si consideri il seguente programma, dove la funzione `som()` agisce direttamente sulla variabile globale. Osservando semplicemente il programma principale, questo non può essere compreso. Pertanto si possono creare problemi quando il richiamo di una funzione provoca modifiche inaspettate su variabili che non le sono state passate come parametri.

```
int N;               /* variabile globale */
.........
int som(int j)
  { return N = N + j; }
.........
void main(void)
{
int i, ris;

N = 9; i = 1;

while (i < N)
  {
  ris = som(i);
  printf("%d - N = %d ris = %d\n", i++, N, ris);
  }
```

```
}
```

Questo e' confermato dalla seguente traccia di esecuzione:

```
1 - N = 10 ris = 10
2 - N = 12 ris = 12
3 - N = 15 ris = 15
.........
48 - N = 1185 ris = 1185
49 - N = 1234 ris = 1234
.........
255 - N = 32649 ris = 32649
256 - N = -32631 ris = -32631
```

10.15 Sottoprogrammi: mancata Inizializzazione

Come ogni variabile, anche quelle globali devono essere inizializzate. Per queste, l'attenzione deve essere maggiore poiché essendo poste fuori da procedure e' facile dimenticasi di inizializzarle anche se l'inizializzazione di variabili globali può essere effettuata insieme alla loro dichiarazione.

```
int N;                /* variabile globale */
int som(int j) { return N += j; }
……….

void main(void)
{
int ris, i; /* N = 1;  */

for(i = 1; i < 10; i++)
  { ris = som(i);
  printf("%d - %d\n", i, ris);
  }
}
```

Questo e' confermato dalla seguente traccia di esecuzione:

```
1 - 11
2 - 13
3 - 16
4 - 20
.......
```

Molti compilatori C effettuano per convenzione l'inizializzazione a ZERO delle variabili globali non inizializzate dall'utente. Questo comportamento non e' standard e pertanto non e' opportuno farvi affidamento.

10.16 Sottoprogrammi: fallimento della proprietà commutativa

In questo paragrafo viene proposto un altro problema derivato dall'uso delle variabili globali. Con l'assegnazione `somma = f1 + f2`; si intende usualmente effettuare la somma di due variabili o funzioni. In questo caso le funzioni `f1()` e `f2()` agiscono sulla stessa variabile globale `N`, ma in modo diverso. Anche se il programmatore e' a conoscenza del comportamento esatto di tali funzioni nei riguardi della variabile globale, può non essere a conoscenza che i compilatori eseguono le operazioni che godono della proprietà commutativa secondo una loro convenzione.
Talvolta possono influire anche i parametri di ottimizzazione del compilatore stesso, che in certi casi possono essere variati dall'utente. In C vi sono altri problemi poiché usualmente i compilatori C sono ottimizzati, cioè hanno dei meccanismi per analizzare il codice al fine di renderlo più efficiente dal punto di vista del tempo di esecuzione e dell'uso memoria.

```
int N;                /* variabile globale */
int f1(void) { printf(" f1 "); return N *= 10; }
int f2(void) { printf(" f2 "); return N += 10; }

void main(void)
{
int somma;
N = 1;
somma = f1() + f2();          /* procedimento errato */

/* somma = f1();
somma += f2();                /* procedimento corretto */

printf("1 - f1 + f2 = %d\n", somma);
N = 1;

somma = f2() + f1();          /* procedimento errato */
/* somma = f2();
somma += f1();                /* procedimento corretto */

printf("2 - f2 + f1 = %d\n", somma);
}
```

Utilizzando la versione precedente, il compilatore ha ottimizzato l'uso delle chiamate alle funzioni `f1()` e `f2()` che risultano in entrambi i casi chiamate nello stesso ordine. Pertanto il compilatore ha ritenuto opportuno eseguire la prima da sinistra a destra e la seconda da destra a sinistra. Come si nota dalla seguente traccia il risultato dipende dall'ordine con il quale vengono eseguite le procedure:

```
f2  f1 1 - f1 + f2 = 121
f1  f2 2 - f2 + f1 = 30
```

Pertanto per risolvere il problema e' opportuno utilizzare il procedimento alternativo che nel programma precedente e' stato commentato. In tale modo il comportamento del compilatore non influisce sul risultato.

Pertanto, e' necessario utilizzare forme sintattiche più esplicite affinché non ci sia una diversa interpretazione da parte di compilatori diversi.

Il problema viene corretto decommentando il procedimento corretto in giallo e commentando quello errato.

10.17 Sottoprogrammi aperti e chiusi

La chiamata di un sottoprogramma altera la normale esecuzione sequenziale del programma passando l'esecuzione al codice realizzato nel blocco del sottoprogramma; alla sua conclusione riprende l'esecuzione del programma dal punto in cui era stata interrotta. La conclusione della chiamata in genere coincide con l'ultima istruzione del sottoprogramma invocato.

I sottoprogrammi realizzati applicando le regole descritte in precedenza sono anche detti sottoprogrammi *chiusi*, perché determinano ogni volta il passaggio del controllo a un gruppo d'istruzioni memorizzate in una determinata area. Esistono altri tipi di sottoprogrammi, detti *aperti*, per i quali in corrispondenza della loro chiamata il compilatore inserisce una replica delle linee di codice che svolgono la funzionalità richiesta. In questo modo durante l'esecuzione non si ha il passaggio al sottoprogramma e quindi si evita il cambio di contesto. In definitiva, ogni volta che si presenta un sottoprogramma aperto il compilatore realizza l'espansione diretta del codice sviluppando il sottoprogramma aperto con i riferimenti ai parametri attuali, senza che sia necessario alcun salto.

In C alcune delle funzionalità standard sono ottenute tramite sottoprogrammi aperti. Il programmatore può inoltre scrivere sottoprogrammi aperti per mezzo della sintassi delle *macro* provenienti dal C o del costrutto `inline` del C++.

10.18 Le macro del C/C++

Una macro e' una porzione di codice realizzata attraverso l'istruzione `#define`, a cui e' associato un identificatore. In fase di preprocessazione, all'identificatore viene sostituito il codice ad esso corrispondente. L'istruzione `#define` e' già stata impiegata per la definizione di simboli a cui sono associati valori costanti. E' possibile, tuttavia, applicare l'istruzione anche ad identificatori con parametri; nell'espandere il codice, il preprocessore ne opera la sostituzione.

Nei seguenti esempi di macro con parametri, la prima serve a determinare il maggiore di due operandi e la seconda il minore.

```
   /* macro che determina il maggiore di due elementi      */
#define max(_op1, _op2) (((_op1) > (_op2)) ? (_op1) : (_op2))
   /* macro che determina il minore di due elementi        */
#define min(_op1, _op2) (((_op1) < (_op2)) ? (_op1) : (_op2))
```

Si noti che in questo caso e' stato utilizzato un particolare costrutto di selezione del C. Si veda la sezione 5.11.

In seguito e' mostrato come vengono invocate le macro. La modalità di chiamata e' esattamente quella dei sottoprogrammi chiusi.

```
int a, b, c;
char A, B, C;
.......
c = max(a, b);          // chiamata della macro max()
C = min(A, B);          // chiamata della macro min()
.........
```

Dopo l'esecuzione del preprocessore, il codice delle macro viene espanso nel modo seguente:

```
c = (((a) > (b)) ? (a) : (b));
C = (((A) < (B)) ? (A) : (B));
```

Al secondo membro delle assegnazioni ci sono delle espressioni condizionali in cui sono stati sostituiti i parametri. Come si può notare le macro sono più flessibili dei sottoprogrammi chiusi, infatti `min()` e `max()` possono essere impiegate correttamente a coppie di dati di tipi diversi. D'altra parte la corretta applicazione di una macro e' tutta a carico del programmatore; nessuna segnalazione viene fornita se una chiamata a macro viene effettuata con parametri di tipo incompatibile con le istruzioni stesse della macro. Nelle macro non esiste la possibilità di definire un prototipo di chiamata e quindi il compilatore non e' in grado di effettuare nessun controllo di congruenza dei tipi dei parametri.

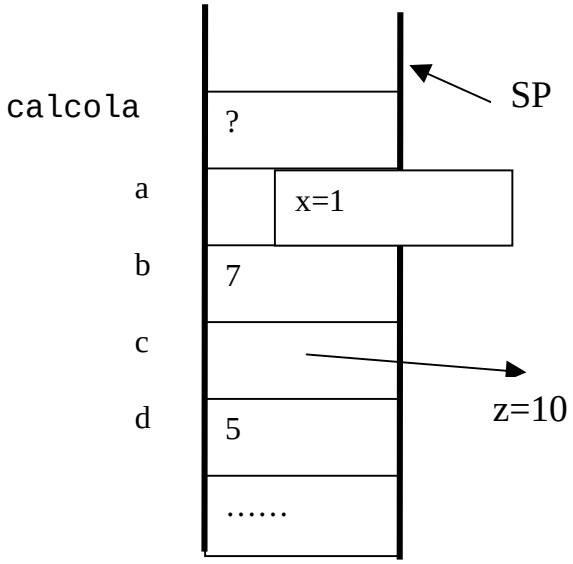
10.19 Lo *stack* e il passaggio dei parametri

Come e' stato accennato durante la presentazione dei sottoprogrammi, quando un sottoprogramma viene eseguito per ogni suo parametro vengono allocate delle celle nella memoria detta *Stack*. Si veda la sezione 1.17. Inoltre vengono allocate nello *stack* anche le variabili temporanee dichiarate nelle procedura ed un cella per il valore di ritorno quando il sottoprogramma e' di tipo *function*.
Tale spazio e' di appropriata dimensione in accordo alla rappresentazione in memoria del tipo di ogni variabile. Lo *stack* e' una particolare area di memoria utilizzata per tale scopo ed e' organizzata come una Pila, secondo la regola FILO, *first in last out*. Il primo che entra e' l'ultimo ad uscire.

Nella funzione dell'esempio seguente si hanno vari parametri passati in vario modo. Al momento dell'esecuzione della funzione `calcola`(), quando l'esecuzione passa al suo interno lo *stack* assume lo stato riportato a fianco:

```
void main (void )
{
int x=1, y=7, z=10, ret=0;
ret=calcola(x,y,&z);
printf("x=%d, y=%d, z=%d, ref=%d\n", x, y, z, ret);
}

int calcola(int & a, int b, int * c)
{
int d=5;
```

```
printf ("calcola0: a=%d, b=%d, *c=%d\n", a, b, *c);
a = a * d ;     b = b * d;      *c = *c * d;
printf ("calcola1: a=%d, b=%d, *c=%d\n", a, b, *c);
stima(b, a, c);
printf ("calcola2: a=%d, b=%d, *c=%d\n", a, b, *c);
return(0);
}
```



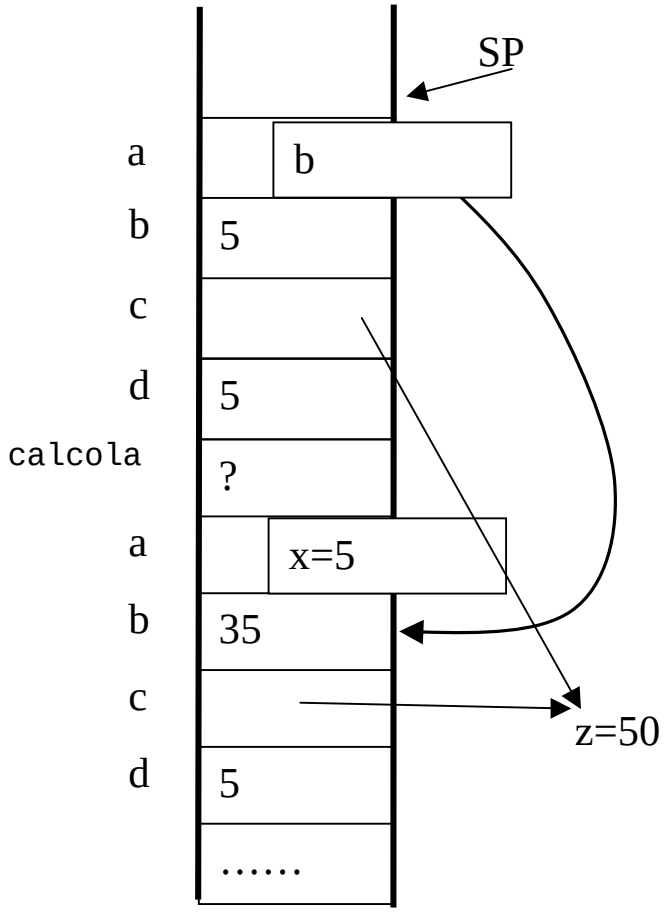
Viene riportato un ? quando il valore non e' stato definito e pertanto dipende dal valore precedente della cella di memoria. In particolare, `a` e' un reference alla cella `x`, `b` e' una copia temporanea del contenuto della cella `y`, `c` e' un puntatore alla cella `z`. In effetti dal punto di vista logico la cella di a può non essere considerata nello *stack*, ma dal punto di vista fisico tale ingombro deve essere considerato visto che per effettuare un effettivo alias alla cella `x` viene utilizzato un puntatore. Si noti che nello *stack* viene inserita anche una cella con il nome stesso della funzione e una cella per le variabili temporanee dichiarate nelle funzione stessa – `d` nel caso specifico. Tale cella viene utilizzata per passare il valore al programma chiamante tramite l'istruzione `return`. Verso il basso vi sono celle di memoria che rappresentano lo stato dello *stack* prima della chiamata.

Le variabili temporanee della procedura sono state inserite nello *stack* in modo sequenziale. Il loro ordine dipende dal linguaggio e dal compilatore pertanto non e' di interesse. Al momento dell'allocazione di nuove variabili nello *stack* viene fatto riferimento allo *Stack* Pointer, SP. Questo riporta l'indirizzo della prima cella libera dello *stack*, anche detta *Top of the Stack*, TOS.
Lo *stack* viene utilizzato con un indice arrovesciato, nel senso che lo SP parte da un valore elevato (pari al massimo numero di celle inseribili nello *stack*, tradotto in byte effettivi) e decresce ad ogni inserimento fino ad arrivare a zero, punto in cui viene segnalato l'errore di *stack overflow* nello *stack*. Tale errore provoca l'arresto dell'esecuzione del programma.

Quando dall'interno di un sottoprogramma si passa a chiamarne un secondo si ha un allungamento dello *stack* verso l'alto dove si vanno ad aggiungere le variabili temporanee del sottoprogramma chiamato.

```
void stima(int & a, int b, int * c)
{
int d=5;
printf ("stima0: a=%d, b=%d, *c=%d\n", a, b, *c);
a = a + d ;     b = b + d;      *c = *c + d;
printf ("stima1: a=%d, b=%d, *c=%d\n", a, b, *c);
}
```

Lo *stack* riportato sulla destra rappresenta lo stato dello *stack* quando si e' all'interno della procedura `stima()` ma non sono stati ancora calcolate le assegnazioni nel punto in cui si stampa `stima1: …...`
Si noti la mancanza nello *stack* della cella per il valore di ritorno della procedura `stima().`



Quando la procedura più interna (`stima`()) e' completata le celle relative alle sue variabili locali sono rilasciate e lo stack si riduce di dimensioni, ritornando alle dimensioni che aveva prima della chiamata. A questo punto anche la procedura (`calcola`()) dalla quale era stata chiamata la procedura (`stima`()) si completa e quindi le corrispondenti celle nello stack vengono liberate.

Di seguito e' riportata la traccia prodotta dall'esecuzione del programma completo.

```
calcola0: a=1, b=7, *c=10
calcola1: a=5, b=35, *c=50
stima0: a=35, b=5, *c=50
stima1: a=40, b=10, *c=55
calcola2: a=5, b=40, *c=55
x=5, y=7, z=55, ref=0
```

Quando si passano molti parametri oppure questi sono di dimensioni considerevoli si deve tenere conto dei costi di allocazione degli oggetti nella memoria *Heap*. Questo aspetto, oltre allo spreco di memoria *Heap*, rende sconsigliabile il passaggio per valore di oggetti di dimensioni considerevoli.

10.20 Criteri di scelta per il passaggio dei parametri

E' stato mostrato in precedenza che i parametri ai sottoprogrammi possono essere passati per valore o per indirizzo (reference, riferimento). Vi e' inoltre la possibilità di passare il valore dell'indirizzo di una variabile e con tale puntatore fare riferimento direttamente alla variabile stessa che e' visibile nel dominio di validità del programma chiamante. Questa ulteriore tecnica viene utilizzata moltissimo in C come surrogato del passaggio di parametri per indirizzo o riferimento. Dato il suo effetto non e' dissimile dal passaggio dei parametri per indirizzo e pertanto può essere considerata una sua variante.

Le modalità di passaggio dei parametri, per valore o per indirizzo, hanno funzionalità diverse e possono essere utilizzate per scopi diversi.
Tipicamente si passano per

- valore i dati che vengono semplicemente utilizzati all'interno di procedure senza che queste abbiamo necessita' di modificarne il contenuto. Sono pertanto variabili di sola lettura per la procedura, cioè di solo ingresso.

- Indirizzo i dati che devono venire modificati dalla procedura stessa e non si ha interesse ad averne una copia aggiuntiva o questa e' già stata fatta in precedenza. Sono pertanto variabili di lettura/scrittura per la procedura, cioè di ingresso/uscita.

Questo tipo di approccio e' troppo semplificativo poiché il passaggio dei parametri per valore implica l'allocazione dinamica di memoria all'interno della memoria destinata allo *Stack*. Allocare dinamicamente variabili nello stack significa anche utilizzare un parte del tempo di CPIU per tale operazione.

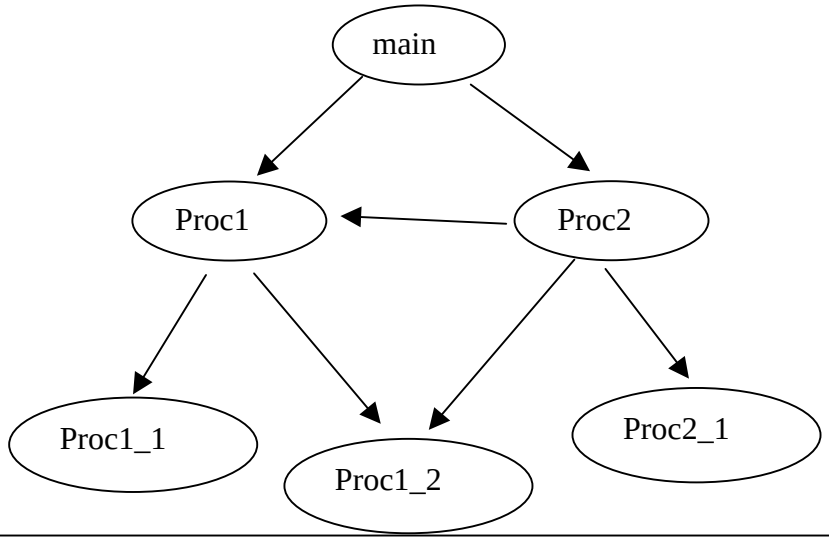
Lo *Stack* e' un porzione di memoria dedicata a tali operazioni organizzata come una pila ma di dimensione limitata. Questa limitazione impone un controllo stretto sulle dimensioni dello *stack* ad ogni chiamata di procedura.

A causa di questa limitazione e' spesso conveniente passare per indirizzo tutte le variabili di una certa dimensione in termini di ingombro in memoria. In questo modo si limita il l'utilizzo dello *stack* solo per memorizzare il puntatore a tali variabili.

10.21 Albero delle chiamate

Si chiama albero delle chiamate un grafo orientato che si diparte da una procedura o programma principale evidenziando le chiamate a sottoprogrammi che da questo si dipartono, considerando le chiamate che si dipartono anche dai sottoprogrammi chiamati fino a raggiungere sottoprogrammi che non contengono altre chiamate. Sono tipicamente esclusi da questi grafi le chiamate a funzioni di libreria.

```
void Proc1(void)
{
Proc1_1();
Proc1_2();
}

void Proc2(void)
{
Proc2_1();
Proc1_2();
Proc1();
}

void main (void)
{
Proc1();
Proc2();
}
```



Chiamate ricorsive (si veda sessione 16) danno luogo a percorsi ciclici nel grafo/albero delle chiamate.

11 Modulo: I tipi strutturati

Per l'organizzazione dei dati e' spesso necessario definire dei nuovi tipi di dati a partire dai tipi elementari e/o dai tipi definiti in precedenza. In C++, e' possibile definire dei dati strutturati specificando sia la struttura dati che le operazioni/funzioni che possono essere utilizzate per manipolarli.

In questo modulo sono presentati i concetti relativi alla realizzazione di nuovi tipi di dati per mezzo della definzione di classi tipica degli approcci object-based (basati sugli oggetti) e object-oriented (orienteati agli oggetti).

In modo semplice si può dire che in questo corso viene proposto un modello di programmazione Object-Based, cioè basato ad oggetti, nel quale non si fa uso dei meccanismi di ereditarietà. Questi saranno

introdotti nel corso di Fondamenti II e sfruttati a pieno nei corsi di programmazione avanzata e/o in quelli di Ingegneria del Software.

11.1 Definizione di tipi di dati

In C++, prima di poter utilizzare un nuovo tipo di dato questo deve essere *definito* e *realizzato*.

Definire un tipo di dato significa descrivere la sua struttura dati interna e il dominio delle operazioni che potranno essere utilizzare per manipolare le istanze di tale nuovo tipo di dato. La classe e' la struttura definizionale attraverso la quale si possono definire nuovi tipi di dati. Da questi nuovi tipi di dati si possono dichiarare delle variabili che vengono dette istanze od oggetti.

Realizzare un nuovo tipo di dato significa implementare le funzioni/operazioni che potranno essere utilizzate per manipolare le istanze di tale nuovo tipo di dato.
Il C++ e' un linguaggio orientato agli oggetti. In questo corso non sono sfruttate tutte le caratteristiche del linguaggio che lo rendono conforme con il paradigma di modellazione ad oggetti, OOP (Object Oriented Paradigm). E' necessario infatti, prima di passare alle questioni dell'object-oriented conoscere bene i fondamenti della programmazione.

Pertanto, il C++ per come viene introdotto ed utilizzato in questo corso deve essere considerato come una via di mezzo fra il C ed il C++. Una sorta di C con una gestione più raffinata e potente per la realizzazione di nuovi tipi di dati. Molto simile al concetto di *Abstract Data Type, ADT*, che vede il tipo di dato come un modello che permette di produrre variabili complesse senza sapere come queste sono internamente strutturate ma conoscendone solo le operazioni che possono essere utilizzate per manipolare le variabili di tale tipo.

11.2 La Classe: definizione di nuovi tipi di dati

Per la definizione di nuovi tipi di dati si utilizza il costrutto `class` del C++. Una nuova classe viene definita tramite la seguente struttura:

```
class <Nome della classe>
          [ : [public | protected | private ]
               <nome della superclasse> ]
{
     private:
          // Attributi e metodi che sono accessibili
          // solo dalla classe in cui sono definiti.
     protected:
          // Attributi e metodi non accessibili
          // dall'esterno ma solo dalle classi derivate
     public:
        // Metodi visibili dall'esterno (secondo
        // l'OOP    non dovrebbero esistere attributi pubblici)
};
```

La classe e' definita elencando i suoi membri. I membri di una classe possono essere variabili, cioè componenti oppure funzioni/operazioni per manipolare le istanze della classe. I membri della classe sono

detti attributi quando sono variabili mentre vengono detti metodi quando sono procedure o funzioni. Un attributo di una classe apparterrà ad un tipo base oppure ad un'altra classe. Con tale definizione si viene a definire una relazione formale di aggregazione fra due tipi o due classi, si ha pertanto un legame is-part-of, e' parte di.

11.3 La definizione di Classe: categorie dei membri

Tali membri della classe possono essere specificati come appartenenti a tre categorie distinte:

- **private:** attributi e metodi che sono accessibili solo dalla classe in cui sono definiti. Tali membri non sono utilizzabili da istanze di altre classi.

- **protected:** attributi e metodi non accessibili dall'esterno ma solo dalle classi derivate. Gli aspetti relativi alle classi derivate non sono considerati in questo corso perché troppo avanzati per programmatori alle prime armi.

- **public:** attributi e metodi visibili dall'esterno (secondo l'OOP non devono esistere attributi pubblici). Si consiglia perciò di definire solamente metodi pubblici e mai attributi.

11.4 Esempio di definizione di una classe, la classe Contatore

In questa sessione viene presentato un esempio di definizione di una classe: la classe contatore. La classe contatore e' definita come una struttura dati avente un valore massimo e un valore attuale. Inoltre il contatore ha delle funzioni che permettono il suo utilizzo: incremento, decremento, leggerne il valore, impostare il valore, cancellare il contatore, etc. Queste funzionalità devono essere implementate con specifici metodi.

```
class Contatore //definizione della classe
{
protected: //attributi e metodi protected
     long val;
     long max;

public: //metodi pubblici
     Contatore();        //Costruttore
     Contatore(Contatore &);       //Clonazione
     Contatore(Contatore *);       //Clonazione per valore
     Contatore(long v, long m);    //Costruttore
     int increment(void);     //incremento
     int decrement(void);     //decremento
     void clear(void);        //azzero il contatore
     long get(void);          //leggo il contatore
     void set(long v,long m=MAXLONG);//preset del contatore
};
```

11.5 Utilizzo della classe Contatore

Sulla base di tale definizione si può ipotizzare l'uso di istanze di questa classe. Per esempio nel seguente spezzone di codice sono stati dichiarati due contatori con diversi valori iniziali:

```
Contatore c1, c2, c3;
.........
c1.set(0, 1000); // contatore che parte da 0 e va fino a 1000
c2.set(10, 100); // contatore che parte da 10 e va fino a 100
c3.set(10);// contatore che parte da 10 e va fino al max della
           // dinamica dei long come valore di default
.........
c1.increment();  // incremento del contatore c1
c2.clear();      // cancellazione del contatore c2
```

Si noti la sintassi per applicare un metodo ad un oggetto della sua classe.
E' inoltre interessante vedere come e' stato definito il valore di default per il valore massimo del metodo `set()`. Questo entra in gioco quando tale metodo viene invocato con un parametro solo, come nel caso del contatore `c3`.
Questo e' ovviamente possibile solo per gli oggetti che sono istanziati dalla classe corrispondente.
Nello stesso modo si possono dichiarare degli *array* di oggetti di una certa classe:

```
Contatore matcont[10][10]; // matrice di contatori
```

Nella classe `Contatore` come in tutte le classi C++, i metodi possono essere divisi in metodi costruttori, distruttori, selettori, operatori e funzionali. Nel precedente esempio i metodi `set()` e `get()` sono metodi selettori perché permettono di accedere direttamente ai valori degli attributi della classe impostando o restituendone il loro valore. I metodi `increment()`, `decrement()` e `clear()` sono metodi funzionali poiché rappresentano le funzionalità della classe. Per i metodi costruttori, distruttori ed operatori si veda in seguito.

Si possono anche definire dei puntatori a variabili di tipo `Contatore`. Questi possono essere inoltre utilizzati per referenziare `Contatori` allocati staticamente con dichiarazioni oppure per allocare dinamicamente nella memoria *Heap* dei contatori:

```
Contatore *cf, *cg, c3;
.............
cf= &c3;         // assegnazione dell'indirizzo di c3 a cf
cg = new Counter;   // allocazione dinamica
```

11.6 Metodi Costruttori (classe Contatore)

Fra i metodi della classe, i costruttori rivestono una particolare importanza perché specificano cosa si deve fare al momento della istanziazione di oggetti della classe. Si ricorda che il processo di istanziazione può essere dinamica o statica. Lo stesso costruttore può essere utilizzato per effettuare entrambi i tipi di istanziazione.

I costruttori possono essere facilmente riconosciuti poiché il loro nome coincide con quello della classe. Nella definizione e realizzazione della classe si possono definire svariati metodi costruttori purché questi si differenzino per numero o tipo di parametri. Questo approccio che consente la definizione e realizzazione multipla di metodi si chiama *overloading*. Nella classe `Contatore` sono stati definiti due metodi costruttori. Il primo senza parametri che costituisce il costruttore di *default*:

```
Contatore::Contatore() : val(0), max (0) {}
```

I valori di *default* sono specificati dopo il segno di ":" con una lista di valori di default. La presenza del costruttore assicura che le istanze abbiano la loro allocazione dei valori iniziali coerenti con la natura del tipo realizzato. Questo elimina i problemi che possono derivare dalla mancata inizializzazione delle variabili/componenti del dato. Anche la lista dei valori di default deve far sì che tutti gli attributi della classe siano impostati a dei valori iniziali noti e compatibili con le funzionalità della classe. Questo e' ovviamente compito del programmatore.

Si noti anche che l'implementazione del costruttore implica la scrittura del nome della classe seguito dal nome del metodo separati da ::, questi nel caso del costruttore coincidono, ecco perché si ha `Contatore::Contatore`.

Tale costruttore viene eseguito tutte le volte che si opera un istanziazione di una variabile Contatore senza fornire dei parametri iniziali, come ad esempio in: `Contatore cx, cy;` Questo significa che i contatori `cx` e `cy` hanno i loro rispettivi valori di `val` e `max` posti a 0. Il secondo costruttore:

```
Contatore::Contatore(long v, long m) { val=v; max=m; }
```

permette di imporre il valore di tali parametri interni direttamene al momento della dichiarazione del contatore:

```
Contatore cx(0,100);
```

Lo stesso costruttore viene utilizzato quando si ha un'allocazione (istanziazione) dinamica per mezzo dell'istruzione `new`:

```
Contatore *cg;
.............
cg = new Counter(10, 100); // alloc. dinamica con parametri
```

11.7 Implementazione di metodi (classe Contatore)

Il codice seguente riporta l'implementazione degli altri metodi della classe contatore. Si noti che l'implementazione dei metodi implichi il dichiarare a che classe fanno parte. Questo viene effettuato dichiarando prima del nome del metodo il nome della classe seguito dal grafema "::".
La realizzazione dei metodi se non e' realizzata nello stesso file in cui si trova la dichiarazione della classe deve essere preceduta dall'inclusione del file che contiene la definizione della classe. Questa e' indispensabile affinché i metodi possano essere definiti ed assegnati alla classe stessa. Il compilatore

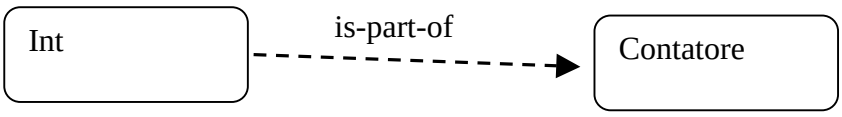
effettua un controllo stretto fra i metodi dichiarati e quelli realizzati in modo da verificarne l'effettiva realizzazione e la coerenza in termini di tipi di dati dei parametri. Per tutti i metodi della classe e' possibile definire dei metodi omonimi ma che differiscano per i parametri. Tale tecnica di *overloading* non e' limitata ai costruttori della classe.

```
#include "Contatore.hxx"

int Contatore::increment(void)
  {  if(val>=max) return ERROR;
  val++;
  return OK;
  }

int Contatore::decrement(void)
  {  if(val==0) return ERROR;
  val--;
  return OK;
  }

void Contatore::clear(void) { val=0;}

long Contatore::get(void) { return val;}

void Contatore::set(long v, long m)
   { val = v;       max = m; }
```

E' importante notare come dai metodi della classe Contatore sia possibile accedere direttamente agli attributi della classe. Questo significa che il campo di visibilità degli attributi della classe si estende anche a tutti metodi anche se questi non sono implementati dentro lo stesso file in cui e' definita la classe, e non sono implementati dentro la coppia di parentesi graffe che definisce la classe.

11.8 La notazione punto per i membri della classe

Quando si hanno dei tipi strutturati come le classi queste sono delle strutture dati che hanno al loro interno delle componenti, degli attributi: istante di altre classi o tipi base. Tale legame, fra la classe che descrive l'istanza utilizzata e la classe che la usa e' una relazione di aggregazione detta di IS-PART-OF. Per esempio, considerando la classe `Contatore` precedentemente vista questa contiene due interi. Fra la classe degli interi e la classe contatore vi e' un legame di IS-PART-OF. Gli interi sono parte del contatore che si rappresenta come segue.



La notazione punto per accedere alle componenti viene utilizzata per indicare tutti i membri della classe inclusi i metodi stessi e quindi le chiamate. Questa notazione ha la seguente sintassi BNF:

```
<oggetto di tipo X> . <membro della classe X>
```

In C/C++ per accedere alle componenti di una struttura dati si può utilizzare la notazione punto; detta *dot notation*. Per esempio vediamo la realizzazione del costruttore per la clonazione di un contatore:

```
Contatore::Contatore(Contatore & a) { val = a.val;  max = a.max; }
```

In questo caso, si accede alle componenti del contatore in via di costruzione direttamente con il nome dell'attributo mentre si accede agli attributi del contatore dal quale si opera la clonazione con la notazione punto. Con `a.val` si intende il valore del contatore `a`. L'accesso per mezzo della notazione punto e' permesso solamente dall'interno di metodi della stessa classe. Non e' possibile utilizzare la notazione punto per accedere agli attributi di una classe da altre classi se gli attributi sono stati dichiarati privati o protetti, come e' necessario per non violare il *data hiding* (occultamento dell'informazione).

Secondo questo concetto base della programmazione per ADT e in seguito acquisito dalla programmazione OOP, gli attributi di una classe possono essere manipolati solo dai metodi della classe, nessun altro tipo o dato nel sistema deve avere accesso e/o conoscenza della struttura della classe. La realizzazione di un certo tipo per mezzo di strutture dati e' una questione privata del tipo di dato e non deve essere condivisa con altri. Questa regola evita di creare delle dipendenze strutturali strette fra tipi di dati.

In tali casi per accedere alle componenti e' necessario utilizzare dei metodi selettori.

La notazione punto viene utilizzata anche per effettuare le chiamate ai metodi di una classe.

```
Contatore c1;

c1.increment();
…..
c1.set(50);
```

11.9 La notazione redirezione per i membri della classe

Quando non sia ha a disposizione direttamente l'oggetto non e' possibile utilizzare la notazione punto ma si può utilizzare la notazione redirezione. L'operatore di redirezione e' composto da due caratteri -> ed ha la seguente sintassi BNF:

```
<puntatore a un oggetto di tipo X> -> <membro della classe X>
```

Per esempio, si veda l'implementazione del seguente creatore di clonazione che utilizza come parametro il valore del puntatore ad un contatore.

```
Contatore::Contatore(Contatore * a) { val = a->val;  max = a->max; }
```

La stesse operazioni potevano essere effettuate utilizzando la notazione punto e l'operatore * che identifica l'oggetto dato il puntatore:

```
Contatore::Contatore(Contatore * a) { val = (*a).val;  max = (*a).max;
}
```

In questo caso e' necessario utilizzare le parentesi tonde poiché l'operatore punto ha una maggiore priorità rispetto all'operatore * che permette di ottenere l'oggetto dato un puntatore. Pertanto `(*a).val` ha un senso mentre `*a.val` e' sintatticamente sbagliato.

Tale notazione si utilizza anche per chiamare i metodi qualora si abbia a disposizione il puntatore ad un oggetto e non l'oggetto stesso:

```
Contatore *c1;

c1 = new Contatore(0, 100); // costruzione e inizializzazione

c1->increment(); // incremento
c1->decrement(); // decremento
```

La stesse operazioni potevano essere effettuate utilizzando la notazione punto e l'operatore * che identifica l'oggetto dato il puntatore:

```
(*c1).increment(); // incremento
(*c1).decrement(); // decremento
```

In questo caso e' necessario utilizzare le parentesi tonde poiché l'operatore punto ha una maggiore priorità rispetto a quella dell'operatore * che permette di ottenere l'oggetto dato il puntatore.

11.10 Il puntatore a se stesso degli oggetti, la parola chiave `this`

In alcuni casi e' utile poter conoscere il puntatore ad un certo oggetto (istanza di una classe) direttamente durante lo svolgimento dei metodi stessi della classe. Questo tipo di conoscenza e' diversa dal conoscere l'indirizzo di una variabile quando la si utilizza in espressioni o assegnazioni. Si ricorda che data una variabile e' possibile ottenere il suo indirizzo semplicemente utilizzando l'operatore `&`. Infatti, `&a` e' l'indirizzo della variabile `a`.

In C++, e' possibile fare riferimento al puntatore di un oggetto dal metodo stesso che e' stato invocato per tale oggetto per mezzo della parola chiave `this`. Con il metodo che segue viene chiesto a un contatore di produrre il suo puntatore:

```
Contatore * Contatore::dammiptr(void) { return(this); }
……..
Contatore *c1, c5;

c1 = c5.dammiptr();
```

In questo modo in puntatore `c1` punta al puntatore `c5`.

In seguito sarà utilizzato per la realizzazione di metodi di clonazione, assegnazione e conversione, metodi ed operatori che devono produrre come valore di ritorno un oggetto della stessa classe del metodo.

11.11 Esempio della classe Point

Nel seguente esempio e' stata realizzata una classe per rappresentare i punti. Questa classe realizza il concetto di punto in uno spazio bidimensionale con due coordinate reali.

```
class  Point
{
private:
     double x,y;    // Coordinate
public:
     Point();       //Costruttore di default
     Point( Point &altro ); // Costruttore di copia
    Point( double xx, double yy); //Costruttore con argomenti
// metodi funzionali
    double get_x(); // restituisce x
    double get_y(); // restituisce y
    void set(double xx,double yy); // impostazione coordinate
// Operatori di Point
    Point operator-();
    Point operator=(Point &a);
    Point operator+=( Point &a);
    int operator==(Point &a);
    int operator!=(Point &a);
    Point operator+(Point &a);
    Point operator-(Point &a);
};
```

Per questa classe sono stati definiti molti metodi. Alcuni sono costruttori, altri sono metodi funzionali, ma la maggior parte sono operatori.

11.12 Metodi costruttori e selettori della classe Point

Di seguito sono riportate le implementazioni dei metodi costruttori e funzionali della classe `Point`.

I costruttori permettono sia l'istanziazione semplice senza parametri che quella con parametri. Vi e' inoltre un costruttore che permette l'istanziazione di punti inizializzando il valore del punto con un altro punto, una specie di clonazione.

```
Point::Point() : x(0.0),  y(0.0) { }

Point::Point( double xx, double yy ) : x(xx), y(yy) { }

Point::Point( Point &altro ) : x(altro.get_x()),  y(altro.get_y()) { }
```

Questi costruttori presentano la definizione delle liste di default.
Tali costruttori possono essere utilizzati come segue:

```
Point p1, p2(3.0,476.8), p3(p2);
Point *p4, *p5;
……….
p4 = new Point;
p5 = new Point(2.5, 56.7);
```

11.13 Metodi selettori della classe Point

I seguenti metodi sono sostanzialmente dei selettori poiché permettono di impostare e leggere il valori delle componenti della classe:

```
double  Point::get_x() { return x; }

double  Point::get_y() { return y; }

void Point::set(double xx, double yy) { x=xx; y=yy; }
```

11.14 Definizione di metodi operatori

In C, non e' possibile definire il funzionamento degli operatori per nessun tipo di dato definito dall'utente né per i tipi elementari disponibili direttamente. Questo e' dovuto alla limitata attenzione alla definizione di nuovi tipi di dato che era presente in C. Questa limitazione e' stata superata in C++ dove si fa largo uso dei meccanismi per definire e realizzare nuovi tipi di dati in accordo all'OOP.

In C++, la definizione degli operatori e' possibile. Per esempio tutte le volte che si definisce un nuovo tipo di dato per questo gli operatori classici algebrici, di confronto, di assegnazione, conversione implicita di tipo, etc. non possono essere applicati perché non sono definiti per tali tipi di dati. Per esempio se si sono dichiarate due istanze della classe punto non e' possibile assegnare i valori di un punto all'altro tramite un’assegnazione; non e' possibile confrontare i due punti per vedere se sono coincidenti, non e' possibile sommare due punti provocando lo spostamento di uno rispetto all'altro, etc.

In C++, e' possibile definire queste operazioni specificando la semantica di ogni singolo operatore in termini di istruzioni. Si possono pertanto implementare le funzionalità che vi sono dietro ad ogni operatore che viene reso disponibile dal linguaggio per i tipi base. Questo meccanismo di specifica di operatori viene detto *overloading*.

Definendo in modo formale gli operatori vengono anche definiti i tipi per i quali tali operatori sono applicabili. Sulla base di tali descrizioni il compilatore effettua dei controlli sintattici. Inoltre, in mancanza dell'operatore specifico fra due tipi viene segnalato un errore sintattico grave. In ogni classe e' possibile definire in modo diverso lo stesso operatore purché i parametri, cioè gli oggetti coinvolti siano diversi. Per esempio la somma di un punto con un altro punto e' un operatore diverso dalla somma di un

punto con un intero. Queste due operazioni possono avere semantica diversa, come non essere definite in un certo contesto.

Di seguito sono riportate le implementazioni di svariati operatori con relativi esempi nei quali viene chiarito quando ogni operatore viene eseguito in base alla sintassi.

11.15 Operatori Aritmetici (la classe Point)

Gli operatori aritmetici possono essere unari o binari. E' un operatore unario, il cambio di segno; mentre sono operatori binari aritmetici la somma, la differenza, il prodotto, etc.

Operatore di cambio di segno e' un operatore unario che viene chiamato in esecuzione ogni volta che si ha un segno meno davanti al nome di una variabile `Point` e non si opera una sottrazione: `-p2.` Il codice relativo e' molto semplice:

```
Point Point::operator-() { return  Point(-x, -y); }
```

In questo caso, la funzione ritorna un punto con le componenti avente segno invertito. Quindi il cambio di segno non e' permanente per l'istanza per il quale il metodo operatore e' chiamato.

Di seguito sono riportate le implementazioni degli operatori di somma e sottrazione. In questo caso si ha degli operatori binari, pertanto nell'operatore sono coinvolti due oggetti per crearne un terzo temporaneo che contiene il risultato dell'operazione. L'operazione di somma entra in causa quando si ha una somma fra punti. In questo caso la somma provoca uno spostamento del punto pari alle coordinate del secondo punto.

```
Point Point::operator+(Point &a) { return  Point( a.x+x , a.y+y ); }

Point Point::operator-(Point &a) { return  Point( -a.x+x, -a.y+y ); }
```

La specifica dell'operatore + ha per parametro un punto passato per indirizzo (che viene utilizzato nel metodo in sola lettura) mentre l'altro punto e' il punto stesso per il quale e' stato invocato il metodo. Questo permette di realizzare un operatore di assegnazione che non utilizza molte risorse e tempo di CPU per la sua esecuzione. Anche in questo caso l'operatore non modifica i due punti coinvolti nella somma ma semplicemente crea un altro oggetto per porvi il risultato. Quindi la somma `a=b+c` equivale a:

```
a=b.operator+(c);
```

Pertanto l'operatore di somma viene applicato considerando l'oggetto a sinistra mentre quello a destra e' il parametro, un altro oggetto. Questo e' importante da conoscere quando si realizzano operazioni aritmetiche non simmetriche; per esempio il prodotto di una matrice per un vettore viene definito in base alle dimensioni e in un solo verso dipendentemente se il vettore e' colonna o linea.
Con l'operatore somma si possono scrivere operazioni espressioni utilizzando tipi anche molto complessi. In questo caso, sulla base degli operatori presentati è possibile scrivere:

```
- p1 + (p2 – P3).
```

Il primo segno meno e' un cambio di segno, il segno + e' la somma, il secondo segno meno la sottrazione.

11.16 Operatori di assegnazione (la classe Point)

Con la definizione di operatori di operatori di assegnazione si intende proprio la definizione della semantica dell'istruzione di assegnazione. In questo caso l'operatore di assegnazione viene ridefinito con la seguente sintassi:

```
Point Point::operator=(Point &a) { x=a.x; y=a.y;  return (*this); }
```

Si noti l'utilizzo del passaggio del punto alla destra del simbolo di assegnazione per riferimento e non per valore. Questo permette di realizzare un operatore di assegnazione che non utilizza molte risorse e tempo di CPU per la sua esecuzione. Si faccia inoltre attenzione al valore di ritorno dell'operazione di assegnazione. Come tutte le assegnazioni in C/C++, l'assegnazione stessa può essere inserita in espressioni e il suo tipo dipende dal tipo della variabile assegnata, come il suo valore e' pari al valore assegnato alla variabile assegnata (a sinistra dell'operatore di uguale).
Con tale operatore si possono scrivere assegnazioni anche fra oggetti appartenenti a tipi definiti dall'utente:

```
Point p1, p2=px;
…….
p1 = p2;
```

Quando sulla destra dell'operatore si ha come parametro un tipo diverso da quello della classe si realizzano degli operatori di *casting*, conversione implicita.

Si ricorda che in C/C++ sono definiti anche gli operatori di assegnazione con operazione implicita, si veda……. Di seguito e' riportato l'esempio di un operatore di somma con assegnazione. La sua implementazione si basa sull'uso dello stesso operatore a livello di componenti.

```
Point Point::operator+=( Point &a) { x+=a.x; y+=a.y; return (*this); }
```

11.17 Operatori di confronto (la classe Point)

Si possono definire anche gli operatori di confronto.

Di seguito sono riportati gli operatori di confronto per uguale e confronto per diverso.

Questi operatori sono implementati sulla base delle componenti e ritornano un valore vero o falso, che in C/C++ viene riportato come un numero intero. Anche in questo caso e' stato utilizzato il passaggio del secondo operatore per riferimento. Questo permette di realizzare operatori di confronto molto efficienti dal punto di vista dei tempi di esecuzione.

```
int Point::operator==(Point &a) { return ( a.x==x && a.y==y ); }
```

```
int Point::operator!=(Point &a) { return ( a.x!=x || a.y!=y ); }
```

Si possono ovviamente definire anche gli altri operatori di confronto: >, <, >=, <=, assegnandogli una certa semantica per le operazioni fra punti nel caso specifico.

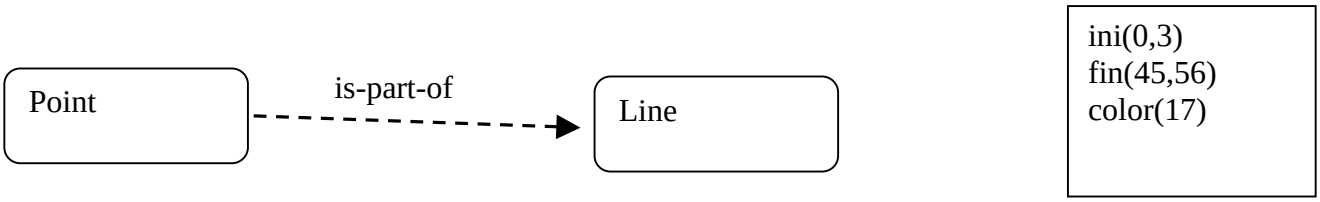
12 Modulo: Classi e relazioni con attributi complessi

Descrizione del modulo da scrivere

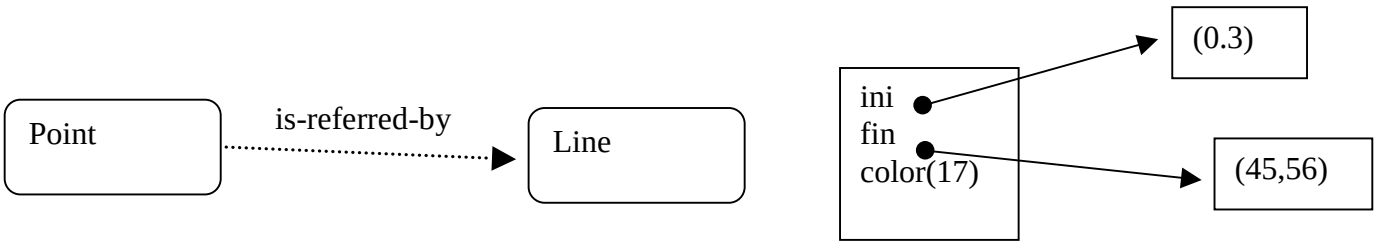
12.1 Rappresentazione delle Relazioni

Fra classi ed oggetti si possono avere due tipi di relazioni:

- **Aggregazioni**. Un'aggregazione fra oggetti si realizza tramite la definizione di un legame is-part-of (e' parte di), come fra punto e linea, un punto e' parte di una linea. In particolare la linea contiene due punti. In questo caso ogni linea ha due punti ed ingombra come due punti piu' un intero. Sulla destra della figura un oggetto della classe Line.



- **Associazioni.** Un'associazione fra oggetti si realizza tramite la definizione di un legame is-referred-by (e' riferito da). Questo legame viene realizzato con un puntatore. In questo caso la linea contiene due puntatori a due punti esterni che possono essere solo delle istanze prodotte in modo indipendente.



12.2 Relazione di Aggregazione: La classe Linea

In questa sessione viene proposto un esempio: la realizzazione della classe linea. La linea viene realizzata come un oggetto composto da due oggetti punto. Il punto iniziale ed il punto finale.

Viene introdotto anche un attributo di colore per la linea.

```
class Line
{
private:
     Point  ini; // punto di applicazione
     Point  fin; // punto finale
```

```
     int color;   // colore del punto
public:
     Line();             // Costruttore di default
     Line(Line &);       // Costruttore di copia
     Line(Point &ini,Point &fin);  //Costruttore con argomenti
     void set(Point &ini, Point &fin, int color);// Impostazione linea
     Point get_ini();         // restituisce ini
     Point get_fin();         // restituisce fin
     int get_color();         // restituisce il colore
// Operatori della classe Line
     int operator==(Line &a);//confronto per uguale nei punti della
linea
};
```

12.3 Construttori della classe linea:

Di seguito sono riportati i metodi costruttori della classe Line.

```
Line::Line() : ini(0,0), fin(0,0), color(0) { }
```

Si noti la presenza della lista di inizializzazione di default `ini(0,0), fin(0,0), color(0)`
Mentre il metodo stesso e' vuoto, non contiene istruzioni specifiche per l'inizializzazione della linea.

Vi potrebbe essere la necessita' di realizzare un costruttore che imposti il colore ad un valore specifico, oppure che inizializzi la linea in base a due punti come il seguente.

```
Line::Line(Point &i, Point &f) : ini(i), fin(f) { }


Line::Line(Line &l):
      ini(l.get_ini()), fin(l.get_fin()), color(l.get_color())
      { }
```

12.4 Metodi selettori per la classe Line

Di seguito sono riportati i metodi selettori della classe `Line`. Questi permettono di accedere in modo indiretto ai valori che caratterizzano la linea. Tali metodi non costituiscono una violazione della regola di *Data Hiding* poiché fornisco delle informazioni non e' detto che nella classe stessa tali informazioni siano nello stesso formato. Per esempio nel caso specifico i punti potrebbero essere stati realizzati nella classe linea come semplici coordinate accoppiate.

```
void Line::set(Point &i, Point &f, int col) { ini=i; fin=f; color=col;
}
```

```
Point Line::get_ini() { return (ini); }  // punto iniziale

Point Line::get_fin() { return (fin); } // punto finale

int Line::get_color() { return color; }  // colore
```

Operatore di confronto per uguale fra linee:

```
int Line::operator==(Line &a) { return( a.ini==ini && a.fin==fin ); }
```

12.5 Programma di prova della classe Line

Di seguito e' riportato un programma principale che può essere utilizzato per collaudare il funzionamento della classe `Line`.

```
int main(void)
{ int j=0;
Line l;
Point p1,p2,p3(100,100);
for(int i=0;i<10;i++)
     {
     p1.set(rand()/100, rand()/100);
     p2.set(rand()/100, rand()/100);
     l.set(p3+p1,p3+p2,(j++)%16);
     }
return 0;
}
```

Il programma di prova definisce 3 punti, questi vengono utilizzati in vario modo: `p3` viene utilizzato come uno spostamento fisso, `p1` e `p2` sono dei punti generati in modo randomico per produrre 10 linee con valori a caso ma spostati nel punto di coordinate `(100,100)`.

12.6 La classe e la modellazione della realtà

Nella fase di analisi di un problema riveste particolare importanza la determinazione delle classi e quindi degli oggetti che vanno a rappresentare le strutture dati che permettono di modellare in modo semplice il problema.

Durante la fase di analisi di un nuovo problema che deve essere risolto per mezzo di un programma vengono evidenziati i vari aspetti del problema in relazione alle informazioni che devono essere manipolate ed agli eventuali algoritmi per manipolarle.
Con la programmazione ad oggetti si tende a replicare con oggetti il mondo degli oggetti reali e quindi delle loro relazioni.

In accordo ai concetti di analisi ad oggetti, le entità del mondo reale sono degli oggetti mentre le classi sono le categorie del mondo reale. Per esempio, se si sta' parlando di un software per *gestire gli spostamenti delle auto di un garage*, le "auto" e' un categoria, mentre la PANDA 750 e' un'istanza della classe auto, e quindi e' un oggetto.

Si può pertanto affermare che in una prima fase l'identificazione delle classi e degli oggetti si opera cercando di identificare nella descrizione del problema quali sono i sostantivi (nell'esempio precedente: auto e garage). Le operazioni per manipolare tali dati possono essere identificate facendo attenzione ai verbi che figurano nella specifica.
Dopo questa fase e' inoltre importante andare a definire le relazioni fra classi ed oggetti nel sistema. La loro organizzazione spesso determina la complessità delle operazioni principali del sistema.

12.7 L'individuazione delle classi ed oggetti

Si identifichi una struttura dati adatta a modellare il seguente problema.

In un'azienda si desidera tenere traccia di tutte le auto che entrano ed escono dal cancello di accesso. Le auto possono essere di dipendenti oppure di visitatori. In questo secondo caso l'entrata e' consentita solo se la responsabilità dell'auto viene assegnata ad un dipendente dell'azienda (per esempio il portiere o la persona che il visitatore e' venuto a trovare). Per ogni accesso si deve tenere traccia della persona che guida l'auto, delle persone che sono a bordo, della data e dell'ora, della persona (dipendente dell'azienda) che ne e' responsabile.

In rosso sono evidenziati i sostantivi:

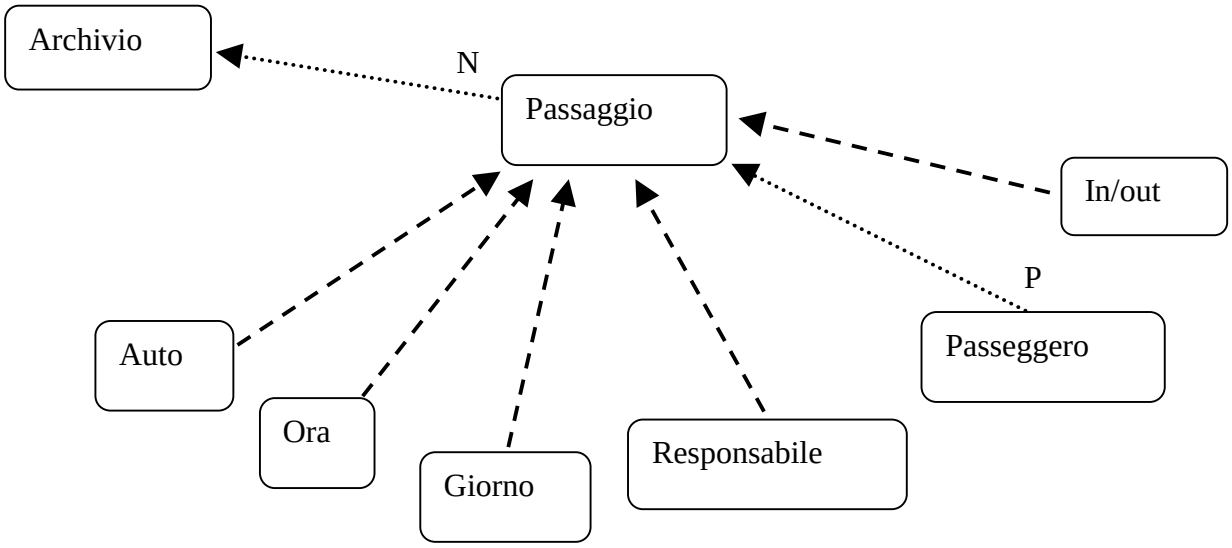
In un'azienda si desidera tenere traccia di tutte le auto che entrano ed escono dal cancello di accesso. Le auto possono essere di dipendenti oppure di visitatori. In questo secondo caso l'entrata e' consentita solo se la responsabilità dell'auto viene assegnata ad un dipendente dell'azienda (per esempio il portiere o la persona che il visitatore e' venuto a trovare). Per ogni accesso si deve tenere traccia della persona che guida l'auto, delle persone che sono a bordo, della data e dell'ora, della persona (dipendente dell'azienda) che ne e' responsabile.

In verde sono evidenziati i verbi che saranno realizzati con metodi:

In un'azienda si desidera tenere traccia di tutte le auto che entrano ed escono dal cancello di accesso. Le auto possono essere di dipendenti oppure di visitatori. In questo secondo caso l'entrata e' consentita solo se la responsabilità dell'auto viene assegnata ad un dipendente

dell'azienda (per esempio il portiere o la persona che il visitatore e' venuto a trovare). Per ogni accesso si deve tenere traccia della persona che guida l'auto, delle persone che sono a bordo, della data e dell'ora, della persona (dipendente dell'azienda) che ne e' responsabile.

La soluzione vede un archivio che tiene traccia dei passaggi. Si ha una relazione 1:N fra Archivio e Passaggio (entrata o uscita). Per ogni passaggio viene memorizzato il tipo di auto, l'ora, il giorno, il responsabile (numero di matricola o nome), se ingresso o uscita (in/out) ed una lista di passeggeri. Questa e' una relazione 1:P fra Passaggio e Passeggero. Tale relazione poteva essere anche stabilita' fra Auto e Passeggero.



12.8 Esempio: la classe Vettore

Nella realizzazione di un vettore si possono definire interessanti operatori che aiutano a controllarne il suo corretto uso. La classe viene definita come segue:

```
class vettore
{
     int dim; // dimensione del vettore
     int * ptr; // puntatore all'area di memoria
                // in cui il vettore e' allocato
public:
     vettore(int); // costruttore con dimensioni
     vettore(const vettore &); // per l'inizializzazione di vettori
                               // alla dichiarazione
     ~vettore();  // distruttore
     int & operator[] (int ); // operatore []
```

```
     void operator=(cost vettore &); // assegnazione
};
```

Nella definizione nella classe si noti la dichiarazione del distruttore e quella dell'operatore []. Il vettore viene realizzato come un oggetto dinamico che può avere dimensioni diverse. Dinamico nel senso che le celle di memoria utilizzate per contenere le componenti del vettore sono allocate nella memoria *Heap* al momento della creazione del vettore in base alle dimensioni imposte al momento della sua creazione.

Sono pertanto possibili per tanto le seguenti operazioni:

```
vettore v1(10), *v2; // vettore di 10 elementi e puntatore a vettore

v2 = new vettore(50);  // allocazioni di un vettore di 50 elementi su
v2
v1[2] = v5[34]; // assegnazione fra interi, uso di componenti

delete v2; // distruzione, dellocazione del vettore v2.
```

L'implementazione dei metodi del vettore e' riportata
di seguito:

```
vettore::vettore(int s)
{
if (s<=0)
  printf("dimensione non valida\n");
dim=s;
ptr=new int[dim];
}
```

Il distruttore libera la memoria *Heap* che e' stata utilizzata.

```
vettore::~vettore() { delete[] ptr; }
```

12.9 Controllo dell'uso degli indici di un vettore (classe Vettore)

L'operatore `[]` e' utile per controllare l'uso degli indici del vettore:

```
int & vettore::operand[](int i)
{
if (i<0 || i>=dim)
   { printf("Errore: indice fuori dalla dimensioni \n");
   return ptr[0];
   }
else return ptr[i];
```

```
}
```

In questo caso, quando viene utilizzato un indice fuori dai valori consentiti in base alla sua allocazione iniziale viene stampato un messaggio e prodotto semplicemente il valore del primo elemento del vettore. Questo operatore viene attivato ogni volta che si utilizzano le `[]` per indicare l'elemento del vettore.

Esercizio:
Lo studente implementi gli operatori che sono stati dichiarati e non riportati e anche gli operatori mancanti di somma, sottrazione e prodotto di un vettore per un numero scalare, stampa del vettore. Si realizzi inoltre anche un programma di prova.

Un ulteriore esercizio consiste nella realizzazione della classe matrice con gli stessi operatori.
Dopo tale realizzazione si lavori sulla realizzazione dell'operatore prodotto vettore matrice. Si effettui i controlli sulle dimensioni della matrice e del vettore per verificare la congruenza dell'operazione.

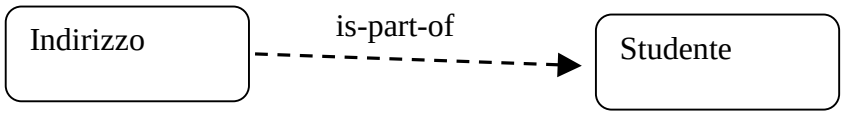
12.10 Una struttura dati informativa composta, la classe Studente

Di seguito e' riportata la definizione della classe studente. Questa presenta diversi attributi definiti in base a tipi elementari ed un attributo che risulta essere un'istanza della classe `Indirizzo`, per rappresentare l'indirizzo dello studente.

```
#define DIMSTR 50
class Studente
{
     char nome[DIMSTR], cognome[DIMSTR];
     Indirizzo indir;  // indirizzo
     int eta;
     char tel[DIMSTR]; // telefono
public:
     Studente(void);
     Studente(char *, char *);
     char * get_nome(void);
     char * get_cognome(void);
     char * get_telefono(void );
     int get_eta(void);
     void set_nome_cognome(char * nome, char * cognome);
     void set_eta_telefono(int, char * telefono);
     void set_indirizzo(Indirizzo & );
     void stampa(void);  // stampa dati
     void getdati(void); // richiesta dati da tastiera
    void modifica(void); // modifica
    Studente operator=(Studente &); // assegnazione
};
```

12.11 Relazione fra Studente e Indirizzo

Si viene pertanto a creare una relazione IS-PART-OF fra la classe Indirizzo e la classe Studente.



Si noti nella definizione della classe la presenza di metodi costruttori e selettori, e del metodo `set_indirizzo()` che accetta come parametro un oggetto indirizzo. E' inoltre definito il metodo `stampa()` che permette di visualizzare a schermo le informazioni contenute nella classe. Ed il metodo `getdati()` che permette di richiedere direttamente i dati all'utente.

```
class Indirizzo
{
     char via[DIMSTR], citta[DIMSTR], loc[DIMSTR];
     int num;
public:
     Indirizzo();
     void getdati(void);
     void getdati(Indirizzo &);
     void stampa(void);
     void set_via(char * via, int numero);
     void set_loc(char * localita, char * citta);
    void modifica(void); // modifica
    Indirizzo operator=(Indirizzo &); // assegnazione
};
```

Come si nota, anche solo guardando le definizioni dei metodi e gli attributi della classe e' possibile capirne il funzionamento di alto livello di un'applicazione software che utilizza tali entità.

12.12 Realizzazione dei metodi della classe Studente

Di seguito sono riportate le realizzazioni dei metodi della classe `Studente` a partire dai costruttori.

```
Studente::Studente(void) { nome[0]=cognome[0]=tel[0]=eta=0; }

Studente::Studente(char *a,char *b)
  { strcpy(nome,a); strcpy(cognome,b); tel[0]=eta=0; }
```

Metodi selettori:

```
char * Studente::get_nome(void) { return nome; }
char * Studente::get_cognome(void) { return cognome ; }
char * Studente::get_telefono(void ) { return tel; }
int Studente::get_eta(void) { return eta ; }

void Studente::set_nome_cognome(char * a, char * b)
  { strcpy(nome,a);  strcpy(cognome,b); }
```

```
void Studente::set_eta_telefono(int a, char * b)
  { eta=a; strcpy(cognome,b); }

void Studente::set_indirizzo(Indirizzo & a) { indir.getdati(a); }
```

Il metodo di stampa della classe studente e' molto semplice, utilizza tutte le informazioni e le presenta a schermo in forma di scheda. Si noti come per la stampa dell'indirizzo sia stata fatta delega all'oggetto di tipo Indirizzo presente nella classe.

```
void Studente::stampa(void)
  { printf ("%s %s \n", nome, cognome);
  indir.stampa();
  printf ("%d anni, tel: %s \n", eta, tel);
  }
```

Metodo per l'acquisizione dei dati da tastiera che sono necessari a riempire le informazioni contenute negli oggetti di tipo Studente.

```
void Studente::getdati(void)
{ printf ("Nome = "); scanf("%s", nome);
printf ("cognome = "); scanf("%s", cognome);
printf ("tel = "); scanf("%s", tel);
printf ("eta' = "); scanf("%d", &eta);
indir.getdati();
}
```

Metodo per la modifica degli attributi di un oggetto studente. Viene presentato il valore corrente a schermo e chiesto di reinserire il valore corretto.

```
void Studente::modifica(void)
{
printf ("Nome = %s : ", nome); scanf("%s", nome);
printf ("cognome = %s : ", cognome); scanf("%s", cognome);
printf ("tel = %s : ", tel); scanf("%s", tel);
printf ("eta' = %d : ", eta); scanf("%d", &eta);
indir.modifica();
}
```

Esercizio:
Un buon esercizio per lo studente e' modificare tale metodo in modo da non modificare il valore corrente se invece di inserire il nuovo valore l'utente semplicemente preme il tasto return/enter.

Operatore di assegnazione/copia fra oggetti di tipo Studente. Con questo metodo si possono effettuare delle copie di oggetti di tipo studente.

```
Studente Studente::operator=(Studente &a)
{
```

```
strcpy(nome, a.nome);   strcpy(cognome, a.cognome);
indir=a.indir;            strcpy(tel, a.tel);
eta= a.eta;               return (*this);
}
```

L'utilizzo della parola chiave `this` permette di assegnare all'istruzione stessa di assegnazione il valore dell'oggetto studente, questo permette di fare assegnazioni concatenate.

```
Studente s1, s2, s3;

s1= s2 = s3;
```

12.13 Realizzazione dei metodi della classe Indirizzo

Metodi costruttori:

```
Indirizzo::Indirizzo() { via[0]=citta[0]=loc[0]=num=0; }
```

Metodi selettori:

```
void Indirizzo::getdati(void)
{ printf ("Via = "); scanf("%s", via);
printf ("Numero = "); scanf("%d", &num);
printf ("Citta = "); scanf("%s", citta);
printf ("Localita' = "); scanf("%s", loc);
}

void Indirizzo::getdati(Indirizzo & a)
{ strcpy(via,a.via); strcpy(citta, a.citta);
  strcpy(loc, a.loc); num=a.num;
}

void Indirizzo::set_via(char * a, int b) { strcpy(via,a); num=b; }

void Indirizzo::set_loc(char *a,char *b) {strcpy(loc,a);
strcpy(citta,b);}
```

Metodo di stampa del singolo indirizzo.

```
void Indirizzo::stampa(void)
{ printf ("%s, %d\n%s, %s\n", via, num, loc, citta); }
```

Metodo per la modifica dell'indirizzo. Viene presentato il valore corrente a schermo e chiesto di reinserire il valore corretto per ogni attributo.

```
void Indirizzo::modifica(void)
{
```

```
printf ("Via = %s : ", via);       scanf("%s", via);
printf ("Numero = %d : ", num);    scanf("%d", &num);
printf ("Citta = %s : ", citta);   scanf("%s", citta);
printf ("Localita' = %s : ", loc); scanf("%s", loc);
}
```

Esercizio:
Un buon esercizio per lo studente e' modificare tale metodo in modo da non modificare il valore corrente se invece di inserire il nuovo valore l'utente semplicemente preme il tasto return/enter.

Operatore di assegnazione fra indirizzi:

```
Indirizzo Indirizzo::operator=(Indirizzo &a)
{
strcpy(via, a.via);   strcpy(loc, a.loc);
num=a.num;            strcpy(citta, a.citta);
return (*this);
}
```

12.14 Collaudo della classe Studenti

Di seguito viene riportato un programma principale per testare il funzionamento delle due classi:

```
void main (void)
{
Studente s1; // dichiarazione di una oggetto studente
Studente s2,s3;

s1.getdati();

printf("----Studente s1 ----");
s1.stampa();

s2=s3=s1;

printf("----Studente s2 ----");
s2.stampa();

printf("----Studente s3 ----");
s3.stampa();
}
```

Il metodo di acquisizione dati per lo studente chiama l'acquisizione dei dati relativa alla classe `Indirizzo`, per l'istanza `indir` contenuta nell'oggetto `s1` di tipo `Studente`.

13 Modulo: La Complessità

Il termine complessità e' utilizzato in informatica per rappresentare diversi concetti. La prima considerazione che possiamo fare è quella di distinguere tra due tipi di complessità:

1) COGNITIVA: complessità nel comprendere un certo algoritmo dal punto di vista dell'essere umano,
2) COMPUTAZIONALE: complessità che valuta un algoritmo per quanto riguarda il calcolatore.

Noi tratteremo solamente la seconda, dato che la prima è soggettiva.

13.1 Complessità degli algoritmi

La complessità di un algoritmo fornisce una misura dell'efficienza dell'algoritmo. Questa dipende dalle risorse che l'algoritmo stesso utilizza. Le risorse che un algoritmo/programma ha a disposizione sono:

1. La CPU, come tempo macchina
2. La Memoria in termini di byte

Il tempo di esecuzione di un algoritmo e' il tempo necessario per svolgere l'algoritmo dalla sua partenza al suo completamento. Tale tempo di esecuzione dipende dal numero di istruzioni che l'algoritmo esegue e quindi da come queste sono organizzate. Tipicamente, il tempo di esecuzione dipende dai dati che vengono utilizzati dall'algoritmo ma può anche dipendere da variabili aleatorie. Alcuni di questi fattori determinano solo variazioni marginali del tempo di esecuzione, altri sono determinanti.

13.2 Complessità: parametri determinanti

I parametri che influenzano in modo determinante il tempo di esecuzione di un algoritmo vengono detti parametri determinanti. Si può intendere la complessità C(X) di un algoritmo X, come un valore correlato con il tempo di esecuzione dell'algoritmo X in funzione di k parametri rilevanti, con k>=1.Quindi $C(X)=f(h_1,...,h_k)$ con h_i = parametro determinante.

Attraverso la complessità è possibile valutare quanto un algoritmo è efficiente e tra più algoritmi che producono gli stessi risultati qual è il più efficiente.

Prima di trattare la complessità per quanto riguarda i linguaggi di alto livello, facciamo un esempio utilizzando l'Assembly: in questo linguaggio ciascuna istruzione fa compiere al clock un certo numero di colpi; ad esempio il MOVE consiste in 1 colpo, il JUMP 2 e il LOAD 4 . Quindi per valutare il tempo di esecuzione e quindi la complessità di un algoritmo in tale linguaggio basta contare il numero dei colpi di clock .

Questo metodo di valutazione non può essere usato per i linguaggi di alto livello, nel quale si misura la complessità calcolando il numero di istruzioni e non quello di colpi di clock.

Per esempio in Assembly le due istruzioni: A : = B*C ; B: = A+9*B
hanno costi diversi nel caso che tali variabili siano in memoria (17 colpi di clock), oppure che siano riferite ai registri (9 colpi di clock). In un linguaggio di alto livello il costo può essere considerato sempre 5 perché non è dato conoscere dove vanno queste istruzioni.

13.3 Prestazioni dei calcolatori

Come e' stato mostrato in precedenza uno dei parametri principali per valutare la complessità di un programma e' valutare il numero di istruzioni che sono da questo eseguite dall'inizio fino al suo completamento.
In modo analogo il numero di operazioni effettuate nell'unita' di tempo e' una misura della potenza di calcolo di un elaboratore. In questa ottica sono utilizzate le seguenti metriche:

- MIPS: milioni di istruzioni Assembly per secondo
- MOPS: milioni di operazioni aritmetiche/logiche per secondo
- MFLOPS: milioni di operazioni in virgola mobile per secondo

I valori ottenuti per queste metriche su elaboratori diversi sono difficilmente paragonabili poiché le singole istruzioni Assembly possono essere molto diverse fra loro. Fra queste la piu' significativa e il MFLOPS che pero' dipende dalla precisione dei numeri in virgola mobile che vengono utilizzati, cioè dipende dal numero di bit di mantissa della rappresentazione interna dell'elaboratore.

Al momento sono valori ragionevoli per un elaboratore i 10 MIPS e il MFLOPScome ordini di grandezza.

13.4 Valutazione della complessità per linguaggi di alto livello

Perché non ci siano indecisioni e per ottenere che la complessità sia indipendente dal sistema di elaborazione, dai dati in ingresso, dal linguaggio e dal traduttore usato, si usa il seguente modello di costo:

1) il costo di ogni operazione di lettura (di una variabile), scrittura (di un dato), assegnazione, confronto e operazione algebrica è unitario,
2) il costo di ogni sequenza di istruzioni è dato dalla somma dei costi delle singole istruzioni,
3) il costo di un ciclo `while` oppure `do-while` è dato dalla somma del costo del test dell'istruzione e del costo di esecuzione delle istruzioni che costituiscono il corpo dell'istruzione iterativa,
4) il costo di un'istruzione `if` è dato dal costo di valutazione della condizione (che è unitario), più il costo dell'esecuzione se la condizione e' vera. Si deve anche considerare il costo delle istruzioni collegate alla parte in alternativa,
5) il costo di una struttura iterativa del tipo `for` è dato dalla somma del costo di inizializzazione della variabile indice (unitario), del costo di incremento della variabile indice, del costo di fine ciclo. Il costo delle ultime due operazioni è unitario,
6) il tempo di esecuzione (costo) di un algoritmo è dato dalla somma dei costi di tutte le istruzioni che lo compongono trascurando il costo di attivazione.
7) Il costo di ogni dichiarazione e' considerato unitario
8) Il costo di ogni chiamata di funzione o procedura pari al numero di parametri del sottoprogramma

In questo modello consideriamo come unico parametro rilevante il coinvolgimento della CPU nell'esecuzione delle istruzioni, cercando di ridurre ad operazioni elementari che hanno un tempo di esecuzione comparabile. Naturalmente gli errori che si compiono sono grandi ma nel contesto di un grosso algoritmo e' possibile farsi un'idea abbastanza precisa del costo computazionale e quindi anche dei tempi di esecuzione quando due algoritmi vengono confrontati.

Per esempio di veda la procedura seguente:

```
void prova(vettore a[], real &y, int s, real t)
{
int i;
y=a[1];
s=1;
for(i=1;i<N;i++)
  { s=s*t;
  y=y+a[i+1]*s
  }
}
```

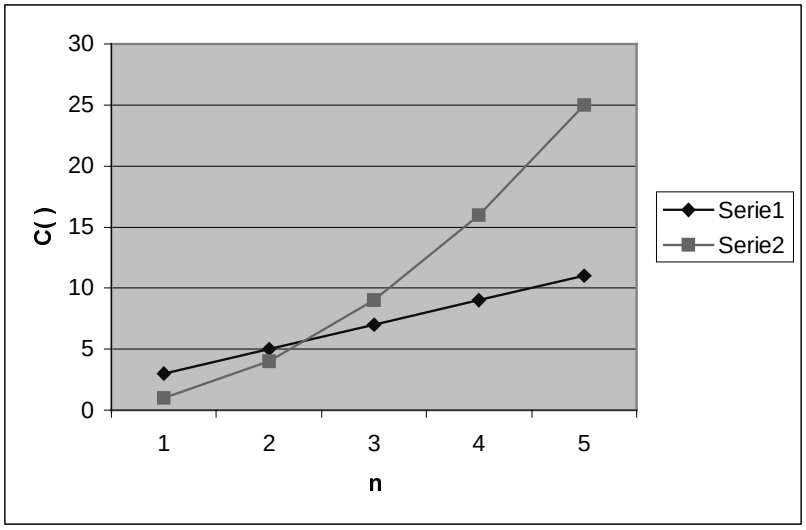
Questo algoritmo ha una complessità pari a C(X)=3+9N. 3 poiché si ha una dichiarazione, e due assegnazioni. 9N poiché per ogni ciclo del `for` si hanno 2 assegnazioni, 4 operazioni nel codice interno e 3 altre operazioni nel corpo dell'istruzione `for`.

13.5 Confronto di algoritmi per complessità

Due algoritmi che risolvono uno stesso tipo di problema sono confrontabili se è possibile esprimere le loro complessità in funzione di alcuni degli stessi parametri determinanti.
Su questa base e' possibile comparare due algoritmi in base alla loro complessità. Se $C(A)<C(B)$ A è migliore di B, se $C(A)>C(B)$, B è migliore di A.
Queste disuguaglianze possono valere solo per certi valori del o dei parametri determinanti. In tale caso e' possibile identificare uno o più valori di taglio. Valori per i quali le due complessità si equivalgono ma che con un incremento di tale valore si ha una complessità maggiore per uno mentre con un decremento si ha una complessità maggiore per l'altro.

Per esempio, se si ha $C(A) = 2n+1$ (serie 2) e $C(B) = n^2$ (serie 1) si ha che per $n<3$, B è migliore di A e che per $n>3$, A è migliore di B.

In generale la complessità di un algoritmo non dipende solo dalla dimensione dei dati, ma anche dalla loro struttura (ad esempio e' intuitivo che la ricerca di un nominativo in un elenco ordinato sia più semplice e quindi meno complessa che la ricerca in una lista non ordinata).



13.6 Complessità: casi migliore, medio e peggiore

Per non complicare l'analisi e poter tenere conto della variabilità del comportamento degli algoritmi in base ai dati si distinguono tre casi: PEGGIORE, MEDIO, e MIGLIORE

Nella valutazione del caso peggiore (migliore) di esecuzione di un algoritmo si fa riferimento a quei valori per i dati in ingresso, in corrispondenza dei quali il costo di esecuzione risulta il più (meno) elevato tra tutti quelli possibili. In questa valutazione i parametri determinanti sono esclusi dal variare.

Per esempio si veda quanto costa nei tre i casi fare un inserimento di uno studente in una fila di banchi. Si nota bene la differenza tra i tre casi, perché questa operazione dipende dalla lunghezza della fila di banchi, N e dal numero di studenti già posizionati.

- IL CASO PEGGIORE si ha quando per inserire uno studente e' necessario spostare tutti gli altri.
- IL CASO MIGLIORE si ha quando lo studente viene inserito in un posto libero e questo viene raggiunto in un passo.
- IL CASO MEDIO si ha quando solo la meta' degli studenti si deve spostare per fare posto al loro compagno (inserimento centrale, spostamento degli altri studenti da una parte o dall'altra)

Dal punto di vista teorico il caso medio dovrebbe essere dato dalla somma su tutti i casi possibili diviso il numero di casi.

13.7 Complessità asintotica

La presenza di valori di taglio e la difficoltà di trovare la funzione di complessità in alcuni casi rendono difficile la valutazione ed il confronto di algoritmi.
Una soluzione consiste nel considerare la complessità dell'algoritmo come funzione dei suoi parametri determinanti quando il loro valore/ o i loro valori tendono all'infinito. Si analizza in questo modo un comportamento asintotico

Questa semplificazione porta nella maggior parte dei casi a risultati significati anche per il confronto di algoritmi. Quando tale operazione non produce risultati che permettono di distinguere l'algoritmo più efficiente dall'altro e' necessario ritornare a modelli più dettagliati come quello presentato in precedenza.

Per identificare la complessità asintotica si deriva la funzione di complessità di dettaglio con i suoi parametri determinanti quindi si trascurano:

1) costanti moltiplicative ,
2) termini additivi di ordine inferiore .

La COMPLESSITA' ASINTOTICA si indica con O(f(N)) e si dice che un algoritmo ha complessità O(f(N)) se per ogni ingresso di dimensione N esegue un numero di operazioni che è proporzionale a f(N), a meno di costanti additive. In pratica la complessità asintotica è il valore a cui tende asintoticamente l'istruzione dominante/determinante dell'algoritmo.

Nel caso che il numero di operazioni richieste sia costante per ogni ingresso, la complessità dell'algoritmo è costante e si indica con O(1) o con semplicemente 1.
Nell'analisi della complessità si nota l'esistenza di alcune istruzioni dette dominanti quando il costo dipende più da certe istruzioni, che da altre e quando la funzione complessità f(N) è uguale al numero di esecuzioni di tali istruzioni.

Si chiamano istruzioni dominanti quelle che vengono eseguite un elevato numero di volte durante l'esecuzione dell'algoritmo. Tale numero di esecuzioni e' fortemente relazionato con i parametri determinati dell'algoritmo. Per esempio in un algoritmo che presenta molte istruzioni di selezione ma un solo ciclo `for` di N iterazioni, si ha che N e' un parametro determinante e quindi le istruzioni dentro il ciclo `for` sono dominanti poiché vengono eseguite N volte.

Qui di seguito verranno enunciate due regole per valutare la complessità:

1) se un algoritmo A e' composto da k parti $P_1,...,P_k$ eseguite sequenzialmente, allora la complessità C(A) è data dalla parte più costosa.

2) se l'algoritmo A richiede k esecuzioni di un sottoalgoritmo e se $f_i(N)$ è il costo della sua i-esima esecuzione , allora $C(A) = O(\Sigma_i^n{}_{=1} f_i(n))$, si ha pertanto un O(NK).

Per esempio:
$3N^2+5N+2$ ha complessità asintotica $O(N^2)$.

14 Modulo: Navigazione in Oggetti con relazioni di aggregazione 1:N

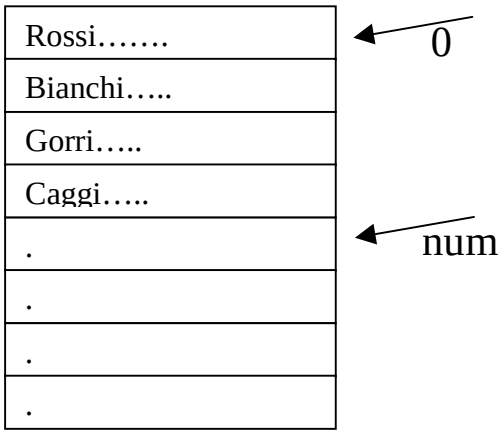
Descrizione del modulo da scrivere

14.1 Un vettore di studenti

Il tipo studente definito in precedenza come la classe studente può essere utilizzato per realizzare una semplice rubrica di indirizzi in memoria centrale. A questo fine può essere utilizzata come struttura dati un vettore di N elementi. Gli N elementi che al massimo possono essere contenuti nella rubrica. Per esempio N=100.

```
Studente vs[100];
```

La gestione del vettore di studenti in termini di inserimento di dati nel vettore può essere realizzata secondo diverse politiche. Per esempio si può effettuare un inserimento sequenziale a partire dall'elemento 0 del vettore mantenere l'informazione relativa al numero di elementi inseriti nel vettore semplicemente utilizzando un indice `num` che identifica il prossimo posto libero.
In figura ogni oggetto inserito e' marcato con il cognome dello studente. Gli oggetti vuoti sono marcati con un *punto.*



In questo modo il vettore di studenti e' organizzato in base all'indice del vettore. Per mezzo di tale indice e' possibile accedere all'informazione dell'oggetto in modo diretto:

```
vs[2].stampa(); // stampa degli elementi dello studente 3
```

Sull'intera rubrica si può effettuare l'operazione di stampa:

```
for (i=0;i<num;i++) vs[i].stampa();
```

14.2 Un vettore di studenti, la ricerca esaustiva

Per ogni oggetto e' possibile identificare un campo chiave, per esempio l'attributo `cognome` della classe `Studente`. Tale campo può essere utilizzato per cercare lo studente fra gli oggetti inseriti in un insieme. Una volta trovato l'oggetto studente ricercato, in base alla `chiave` di ricerca, e' possibile stampare, modificare, etc. le informazioni corrispondenti contenute nell'oggetto.

```
char chiave[20];
```

```
printf("Cognome da cercare: "); scanf("%s", chiave);

for (i=0;i<num;i++)
  { if (strcmp(vs[i].get_nome(), chiave)==0 )
     { // trovato in posizione i-esima
     vs[i].stampa(); // stampa dettagli dell'oggetto trovato
     }
  }
```

Il procedimento di ricerca sul vettore viene detto *esaustivo* poiché tale processo non si interrompe una volta trovato l'oggetto ma continua per tutti gli oggetti che vi sono nel vettore. Vengono pertanto effettuate al limite N operazioni di confronto, sia nel caso di ricerca con successo (che porta a trovare l'oggetto cercato) sia nel caso di ricerca senza successo (che non porta a trovare l'oggetto cercato).

Se all'interno della rubrica sono stati inseriti più nominativi con lo stesso cognome, il procedimento di ricerca esaustiva e' in grado di identificarli tutti e quindi di stamparne i relativi valori (nel caso specifico).

14.3 Un vettore di studenti ordinato, la ricerca esaustiva

Per come viene effettuato l'inserimento, il vettore di studenti si presenta non ordinato, nel senso che gli elementi non sono ordinati in base ad un campo, per esempio al campo chiave; ed in base ad un criterio (per esempio alfabetico decrescente). Nel caso di vettore ordinato e' possibile limitare il numero di confronti a quelli strettamente necessari a trovare l'oggetto in modo lineare, cioè partendo dal primo fino all'oggetto cercato o ad un oggetto che ha un valore di chiave più *pesante* rispetto alla chiave di ricerca. Pesante in questo caso significa successivo o precedente in un ordine alfabetico in base al criterio di ordinamento utilizzato.

Si supponga di avere un vettore di oggetti Studenti ordinato per cognome e si supponga inoltre che il campo chiave (il cognome) identifichi in modo univoco l'oggetto all'interno della struttura.

In queste condizioni e' possibile utilizzare il seguente procedimento di ricerca:

```
for (i=0;i<num;i++)
  {
  if (strcmp(vs[i].get_cognome(), chiave)==0 )
     { // trovato in posizione i-esima
     vs[i].stampa(); // stampa dettagli dell'oggetto trovato
     break;
     }
  }
```

Si noti la presenza dell'istruzione `break` per uscire dal ciclo quando l'oggetto e' stato trovato e stampato. La limitazione riguardo alla presenza di oggetti con la stessa chiave, detti oggetti "duplicati" può essere superata con il seguente codice:

```
for (i=0;i<num;i++)
  { ret= strcmp(vs[i].get_cognome(), chiave);
```

```
  if (ret==0)
     { // trovato in posizione i-esima
     vs[i].stampa(); // stampa dettagli dell'oggetto trovato
     }
  else if (ret<0) break;
  }
```

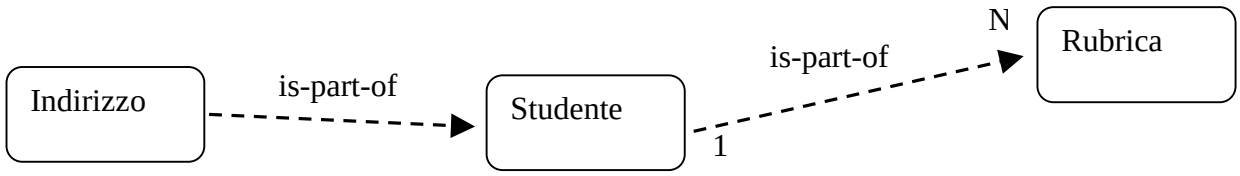
Vengono stampati tutti gli oggetti con chiave identica finche' non si arriva ad un oggetto che ha chiave più pesante, e quindi il procedimento si interrompe, cioè non ci si ferma al primo oggetto con chiave identica. Gli oggetti con chiave identica saranno sicuramente contigui in un vettore ordinato.

14.4 Vettori di oggetti composti, la classe Rubrica (versione vettore)

Si noti che nel caso di un utilizzo pesante della rubrica in termini di numero di procedure o spezzoni di codice che devono essere realizzati per manipolarla, e' opportuno realizzare una specifica classe, per esempio:

```
#define DIMRUB 100

class Rubrica
{
     int num;  // numero degli elementi
     Studente vs[DIMRUB];
public:
     Rubrica(); // costruttore
     void stampa(void); // stampa a video
     Studente * ricerca(char * chiave); // ricerca esaustiva
     int inserimento(Studente &); // inserimento da studente
     ……..
};
```

Con questa definizione di tipo si viene a creare le seguenti relazioni con le classi che sono state definite in precedenza. Si ha pertanto una relazione statica 1:N fra la classe Rubrica e quella Studente, mentre ad ogni studente e' associato un indirizzo, 1:1 viene omesso.



Il metodo `Rubrica::Inserimento()` deve tenere conto delle dimensioni limitate del vettore di studenti, non deve essere possibile eseguire un'istruzione che porti ad inserire dati oltre la cella di indice DIMRUB-1. Tale operazione provocherebbe lo sporcare celle di memoria che non sono adibite al vettore di studenti con il rischio di pregiudicare altri dati e il funzionamento stesso del programma.

Si ricordi che questo e' possibile sotto MSDOS in modalità reale, poiché sia il programma (in termini di istruzioni) che i dati sono allocati nella stessa memoria e il programma stesso può inavvertitamente andare a scrivere nelle celle di memoria adibite a contenere le istruzioni, causando ovviamente la modifica del programma che con grande probabilità può assumere comportamenti imprevisti fino a bloccare l'elaboratore stesso.

In sistemi operativi piu' controllati come Windows 95 o Windows NT tali operazioni sono controllate e impedite.

14.5 Ingombro in Memoria dei dati strutturati

Tutte le volte che si ha a che fare con grandi quantità di dati strutturati, tipi definiti dall'utente, per esempio per mezzo di classi e' opportuno conoscerne la dimensione, cioè l'ingombro in memoria in termini di byte.
Per esempio nel caso della classe Indirizzo, le sue istanze occupano un numero di byte pari a:

```
Dimensione(Indirizzo) = 3*DIMSTR + 4
```

mentre la classe Studente produce oggetti aventi dimensioni:

```
Dimensione (Studente) = 3*DIMSTR + 4 + Dimensione(Indirizzo)
```

Se `DIMSTR==50` si ha che uno Studente occupa 308 byte. A questo punto una rubrica con 100 studenti occupa uno spazio di memoria pari a 30800 byte della memoria di base. Questo spazio di memoria e' occupato anche se vi sono solo pochi elementi nel vettore di studenti. In questo tipo di organizzazione dati l'ingombro in memoria non dipende dal numero di oggetti memorizzati. Se la percentuale degli oggetti utilizzati e' bassa rispetto alle dimensioni totali, questo tipo di organizzazione dati può non essere quella ottimale.

In C/C++, e' possibile utilizzare la funzione `sizeof()` per calcolare la dimensione in byte di un certo tipo di dato. Per esempio e' possibile stimare e stampare le dimensioni degli oggetti prodotti dalla classe studente con:

```
printf ("Classe studente di %d byte \n", sizeof (Studente) );
```

14.6 Esercizio: un vettore di puntatori a studenti

Un approccio diverso per una gestione più dinamica di vettori di oggetti strutturati può essere realizzato definendo un vettore di puntatori a vettori e allocare solo le istanze che sono effettivamente necessarie a partire da tali puntatori:

```
#define DIMRUB  /* numero di elementi nella rubrica */
```

```
Studente *vps[DIMRUB];
```

In questo caso la dimensione del vettore allocato in modo statico dipende dalle dimensioni del puntatore. Per esempio se si ha un puntatore di 2 byte, allora il vettore di 100 puntatori a studente ingombra 200 byte nella memoria di base. Tali puntatori devono essere inizialmente impostati a NULL. In seguito ad ogni inserimento il valore del puntatore corrispondente viene impostato pari all'indirizzo dell'oggetto allocato in memoria *heap*, l'indice `num` (che anche in questo caso indica la prima cella libera del vettore) viene incrementato:

```
vps[num] = New Studente();  //allocazione di uno studente in memoria
heap

vps[num]->getdati ();   // acquisizione dei dati da tastiera

num++; // incremento per indicare il prossimo puntatore vuoto
```

Si noti l'uso dell'operatore di redirezione `->` per chiamare il metodo.

14.7 Vettore di puntatori: la classe RubricaVP (versione vettore di puntatori)

La versione rivista della classe Rubrica utilizzando un vettore di puntatori a studenti non e' molto diversa dalla precedente. In questo caso gli e' stato dato un nome diverso per evitare confusione:

```
#define DIMRUB  100     /* massimo numero di studenti */

class RubricaVP
{
     int num;  // numero degli elementi
     Studente *vsp[DIMRUB];  // vettore di puntatori a studenti
public:
     RubricaVP(); // costruttore
     ~RubricaVP(); // distruttore
     void stampa(void); // stampa a video
     Studente * ricerca(char * chiave); // ricerca esaustiva
     int inserimento(Studente &); // inserimento da studente
};
```

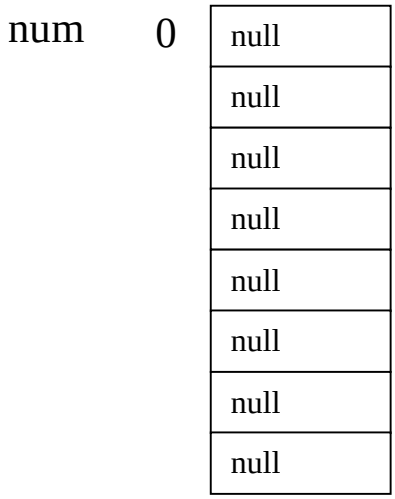
Il metodo `Inserimento()` in questo caso deve tenere conto delle dimensioni limitate del vettore di puntatori a oggetti di tipo `Studente`, non deve essere possibile eseguire un'istruzione che porti ad inserire dati oltre la cella di indice DIMRUB-1.

14.8 Vettore di puntatori alla sua creazione (classe RubricaVP)

Al momento della sua creazione degli oggetti di tipo RubricaVP il vettore di puntatori degli studenti deve essere opportunamente inizializzato. Tale operazione può essere compiuta imponendo un valore nullo ai singoli puntatori per esempio con il seguente costruttore.

```
RubricaVP::RubricaVP()
    { num=0; for (i=0;i<DIMRUB;i++) vsp[i]=NULL; }
```

Pertanto dopo la sua realizzazione la situazione e' quella riportata in figura.



14.9 Vettore di puntatori a regime (classe RubricaVP)

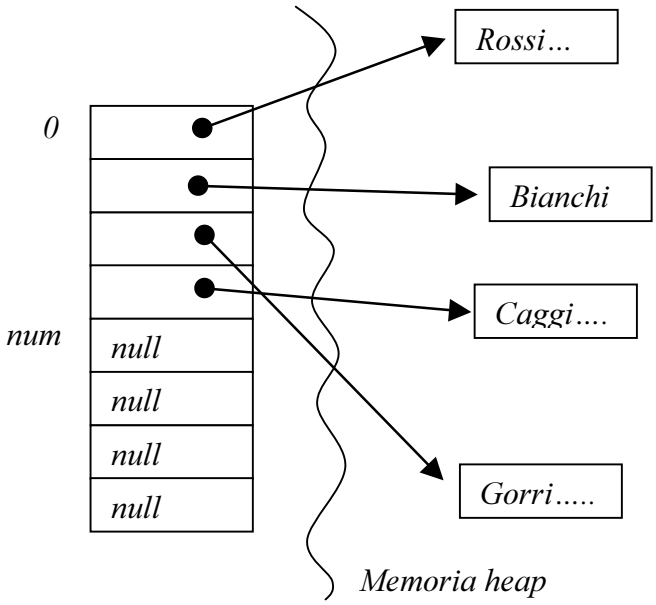
Ad ogni inserimento di un nuovo studente si deve operare un'allocazione come discusso in precedenza. A regime si ha la situazione riportata in figura. Al momento della cancellazione di un elemento e' necessario effettuare anche un'operazione di deallocazione del corrispondente oggetto `Studente` della memoria. Al momento di ogni eventuale distruzione di un oggetto di tipo `RubricaVP` e' necessario eliminare dalla memoria Heap tutti gli Studenti ai quali fa riferimento. Questa operazione può essere effettuato tramite il seguente distruttore:

```
RubricaVP::~RubricaVP ()
  { for (i=0;i<num;i++) delete(vsp[i]); num=0; }
```

Questo metodo viene chiamato tutte le volte che si ha una deallocazione automatica tramite l'uscita da un dominio di validità oppure un deallocazione esplicita dalla memoria *heap* con l'uso della chiamata alla funzione C++ `delete():`

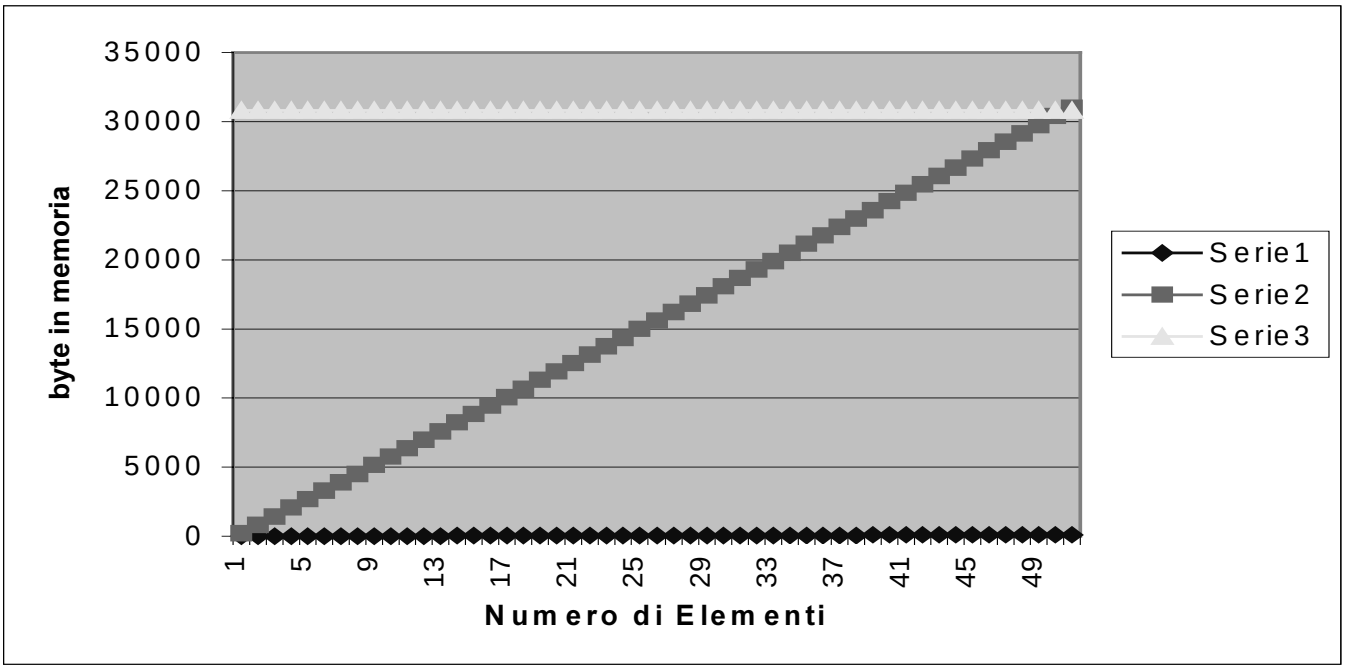
```
Rubrica *rub;
rub = New Rubrica();
```

```
…..
delete(rub);
```



14.10 Confronto fra vettori di oggetti e vettore di puntatori a oggetti

Nel caso di un vettore di puntatori ad oggetti, se si hanno 50 studenti si ha un ingombro di (50 * 308 + 200) byte, pari a 15600 byte. Il seguente grafico riporta l'andamento dell'ingombro in memoria in byte rispetto al numero di Studenti (lungo l'asse X). In giallo le dimensioni della versione con vettore, mentre il lilla la versione con i puntatori. Si noti come sia conveniente la versione basata sui vettori solo quando il vettore e' pieno.



14.11 Metodi distruttori

I metodi distruttori sono realizzati per definire il procedimento di deallocazione dell'eventuale memoria *Heap* allocata durante la creazione di un oggetto. Tali metodi sono molto più importanti dei metodi di allocazione/creazione. Al momento della creazione di un oggetto questo deve ancora inserirsi nell'applicazione stessa e pertanto tali metodi non creano tipicamente problemi. Una volta che un oggetto e' creato, per esempio tramite un'allocazione dinamica si crea un legame fra un puntatore (tipicamente definito come attributo della classe) ed un'area della memoria *Heap*. Tale legame e' esterno alla classe. Altri oggetti nel sistema possono utilizzare il valore del puntatore a tale area di memoria. In questo modo si vengono a realizzare oggetti che condividono un'area di memoria comune.
Questa condivisione e' possibile anche verso oggetti allocati dinamicamente. Per tale motivo l'operazione di cancellazione/deallocazione di tali oggetti e' un momento delicato. Un oggetto potrebbe deallocarli mentre altri oggetti del sistema potrebbero continuare a referenziare tale oggetto con un puntatore. Il rischio e' di utilizzare dati che non hanno più significato oppure cadere in un'eccezione che porta alla conclusione prematura del programma accorgendosi che il programma cerca di accedere ad un'area di memoria non più adibita a variabili del programma.

I metodi distruttori si distinguono dai costruttori per la presenza del carattere '~' (detto tilde) prima del nome. Anche i metodi distruttori hanno come nome il nome stesso della classe. Tali metodi vengono invocati automaticamente ogni qual volta viene utilizzata la funzione `delete` del linguaggio oppure quando si ha una deallocazione in chiusura di una procedura.

14.12 La memoria Heap. Il Garbage Collection

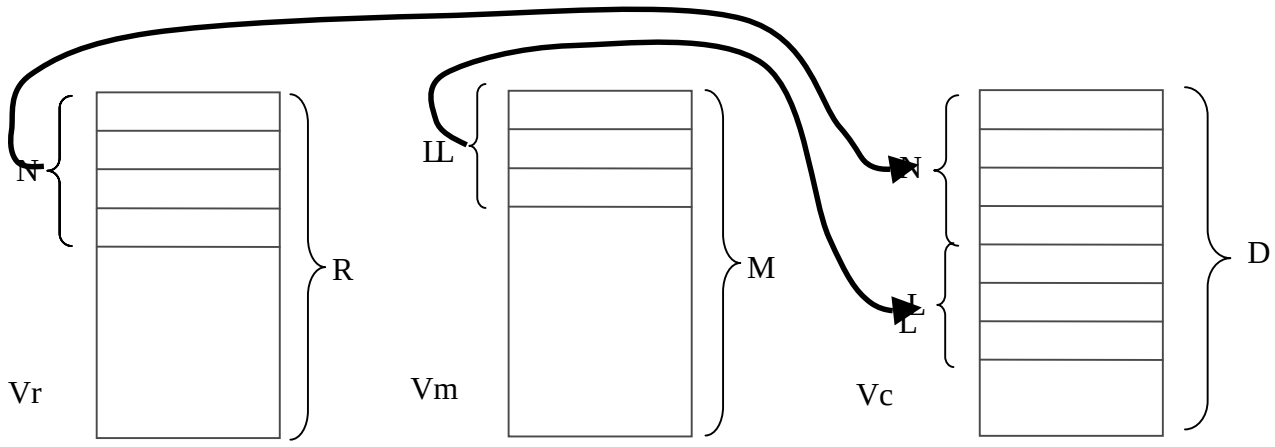
Come e' stato discusso l'uso dei meccanismi di allocazione/deallocazione dinamica (al tempo di esecuzione) sono tipicamente più efficienti per lo sfruttamento della memoria poiché si viene ad utilizzare solo la memoria strettamente necessaria per le variabili. La memoria deallocata viene riutilizzata per allocare altre variabili. Vi e' pertanto un meccanismo che verifica le dimensioni di ogni variabile in corso di allocazione e sceglie dalla memoria lo spazio corretto ove mettere tale variabile.

In applicazioni dove si hanno molteplici allocazioni e deallocazioni di variabili nella memoria *Heap*, questa memoria viene a frammentarsi a causa dello sfruttamento delle stesse aree solo per variabili che hanno un ingombro minore o uguale a quello dello spazio disponibile. Questo meccanismo può portare la memoria ad una degenerazione dell'effetto di frammentazione, che potrebbe portare la memoria a non avere aree consecutive sufficientemente grandi per allocare anche variabili di piccole dimensioni. Al fine di impedire tale degenerazione e' necessario ricompattare la memoria al fine di recuperare i piccoli spazi rimasti fra variabili per formare grandi aree di memoria che possono essere facilmente riutilizzate per variabili di dimensioni significative.

Questa operazione di ricompattamento viene detta di *Garbage Collection*.

14.13 Concatenazione di vettori non ordinati

L'operazione di concatenazione di due vettori (per esempio Vr e Vm) consiste nel prendere le informazioni contenute nei vettori ed inserirle in un terzo vettore di dimensione opportuna. Se il primo vettore ha dimensione R ed il secondo M il terzo vettore deve avere dimensione D pari ad almeno R+M. I due vettori iniziali possono non essere pieni; pertanto se questi hanno rispettivamente solo N ed L elementi il terzo vettore dopo l'operazione di concatenazione avra' N+L elementi.
In figura e' rappresentata l'operazione di concatenazione di Vr a Vm. Questa e' stata realizzata copiando le informazioni di Vm in Vc e quindi copiando le informazioni di Vr di seguito a queste nello stesso vettore Vc.



Si noti che l'operazione di concatenazione non e' simmetrica nel senso che concatenare Vr a Vm e mettere il risultato in Vc e' diverso da concatenare Vm a Vr e mettere il risultato in Vc:

$$Vr \oplus Vm \neq Vm \oplus Vr$$

Dove con $\oplus$ e' stata rappresentata l'operazione di concatenazione. Nell'esempio precedente e' stato fatto:

$$Vc = Vr \oplus Vm$$

La complessità asintotica dell'operazione di concatenazione dipende dalla somma degli elementi delle strutture da concatenare ed al limite dalla somma degli elementi effettivi contenuti nelle strutture, rispettivamente O(M+R) e O(N+L).

14.14 Complessità nella gestione di Vettori non ordinati

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di una certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni.

- Per inserimento si intende l'operazione di inserzione senza considerare la ricerca ma considerando eventuali operazioni di riorganizzazione.

- Per cancellazione si intende la sola operazione di eliminazione dell'oggetto senza considerare la ricerca ma considerando le eventuali operazioni di riorganizzazione.
- Per modifica si intende la sola operazione di modifica dell'oggetto senza considerare la ricerca e senza considerare il caso in cui si ha un cambio chiave e quindi si deve cancellare e reinserire.

Nella seguente tabella sono riportate le complessità asintotica, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su un vettore non ordinato di N oggetti (per esempio un vettore di studenti nella classe `Rubrica` o un vettore di puntatori ad oggetti nella classe `RubricaVP`). Fra queste operazioni l'unica globale e' quella di stampa, le altre sono operazioni sostanzialmente sul singolo oggetto.

Vettore NON ordinato di N elementi				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(1)	1	1	1
Cancellazione	O(N)	1	1	1
Ricerca	O(N)	N/2	1	N
Modifica	O(1)	1	1	1
Stampa	O(N)	N	N	N

15 Modulo: Strutture lineari ordinate

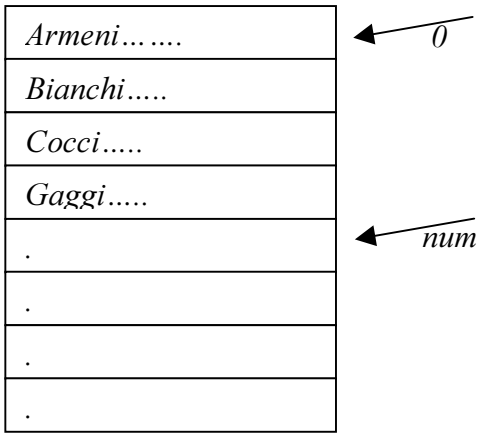
15.1 Struttura dati con Vettore ordinato all'inserzione: la classe Rubrica

In precedenza, e' stata mostrata la classe Rubrica realizzata come un vettore di Studenti. A questo riguardo vengono mostrate le problematiche dell'inserimento ordinato. Per riferimento viene riproposta la classe Rubrica ma tutte le operazioni possono essere effettuate anche con la classe RubricaVP con opportune modifiche.

```
#define DIMRUB 100

class Rubrica
{
    int num;  // numero degli elementi
    Studente vs[DIMRUB];
public:
     Rubrica(); // costruttore
     void stampa(void); // stampa a video
     Studente * ricerca(char * chiave); // ricerca esaustiva
     int inserimento(Studente &); // inserimento da studente
     void cancella(int pos); // cancellazione
     int cancella(Studente *); // cancellazione
     int modifica(Studente *); // modifica
};
```

L'inserimento ordinato di un vettore comporta una prima fase di ricerca della posizione di inserimento e quindi un'operazione di riorganizzazione. Quando si adotta un meccanismo di inserimento ordinato nel

caso generico ci si trova a inserire un elemento in un vettore che contiene già degli elementi. Per esempio, se si considera il vettore ordinato riportato in figura e si ipotizza di inserire lo Studente di cognome/chiave `Caccini` questo deve essere inserito fra `Bianchi` e `Cocci`.



A questo fine e' necessario spostare tutti gli studenti che hanno un valore di chiave maggiore di quello da inserire. Pertanto il metodo di inserimento risulta composto da quattro fasi:

1. ricercare il posto ove l'oggetto deve essere inserito,
2. fare il posto per il nuovo elemento,
3. inserire il nuovo elemento,
4. aggiornare il valore di `num`, il numero degli elementi

Per spostare gli elementi si devono effettuare delle copie. L'operazione di copia, cioè di assegnazione deve essere definita come un operatore della classe Studente, come in effetti e' già stato effettuato in precedenza.

In seguito si adotta l'ipotesi di lavorare con chiavi univoche. Nel senso che la chiave identifica in modo univoco lo studente. Pertanto, l'inserimento di uno studente con un campo chiave identico ad uno studente già presente nella rubrica non deve essere consentito. Questa condizione di allarme può essere verificata effettuando una ricerca.

Tale ipotesi non e' una forte limitazione, visto che qualora si trovi un nuovo studente da inserire con un campo chiave identico vi sono due possibilità:

1. Lo studente in questione e' già stato inserito;

2. Lo studente e' diverso ma uguale per almeno il campo chiave e può essere inserito utilizzando un campo chiave modificato. Per esempio aggiungendo un numero oppure altri caratteri per distinguerlo dal primo.

15.2 Costruttore ed inserimento ordinato: la classe Rubrica

L'operazione di inserimento e' la più critica per la struttura dati in questione. Come e' stato spiegato l'inserimento viene effettuato in tre fasi, il fallimento di tale operazione invalida anche tutte le altre operazioni che hanno per ipotesi la presenza di un vettore ordinato, come la ricerca, successivi inserimenti, la cancellazione, etc.

Oltre a questi aspetti si deve tenere conto che il metodo deve poter garantire:

- l'inserimento del primo elemento quando il vettore e' vuoto. Si veda l'inizializzazione della variabile `pos`.
- L'inserimento dell'ultimo elemento del vettore. Questo caso viene identificato verificando che il `for` di ricerca non ha avuto successo e pertanto `i==num`.
- La presenza di un controllo che impedisca l'inserimento di elementi se il vettore e' pieno.

Di seguito e' riportato il codice della funzione di inserimento nel quale si possono facilmente riconoscere le fasi elencate in precedenza:

```
int Rubrica::inserimento(Studente & astud)
{ int ret, i;

int pos=0; // per garantire l'inserimento del primo
if (num>=DIMRUB) return(1); // Rubrica piena

// ricerca se lo studente e' gia' inserito e della posizione
for (i=0;i<num;i++)
  { ret = strcmp(vs[i].get_cognome(), astud.get_cognome() );
  if (ret==0)
    { printf("Studente gia' presente \n");
    vs[i].stampa();
    return(0);
    }
  else if (ret>0) { pos=i; break; } // posizione trovata
  }
if (i==num) pos=i; // inserimento nell'ultima posizione

// spostamento per fare il posto al nuovo studente
for (i=num-1;i>=pos;i--)
  { // printf ("copio %d in %d \n", i, i+1);
  vs[i+1]=vs[i];
  }
vs[pos]=astud;  // inserimento
num++; // aggiornamento del valore di num
return(0);
}
```

15.3 Stampa del contenuto della struttura dati: la classe Rubrica

La stampa e' un'operazione semplice e consiste nella visita della struttura dal primo all'ultimo elemento. La stampa della rubrica viene delegata alla stampa dei singoli studenti, la stampa del singolo studente prevede la delega della stampa dell'indirizzo ai metodi della classe rispettiva.

```
void Rubrica::stampa(void) // stampa a video
{
int i;

for (i=0;i<num;i++)
  {
  printf("--------Studente numero %d----------\n", i);
  vs[i].stampa(); // stampa dei dati dello studente

  printf("Premi un tasto");

  if (i+1==num)
    printf(" per finire\n");
  else
    printf(" per vedere il prossimo\n");

  getch(); // attesa della pressione di un tasto
  }
}
```

15.4 Ricerca in un vettore ordinato: la classe Rubrica

L'operazione di ricerca e' molto semplice. Questa consiste nell'effettuare un scansione della struttura dati e nel confrontare la chiave di ogni elemento con quella ricercata. Quanto tale operazione ha successo viene ritornato al programma chiamante il puntatore all'oggetto cercato altrimenti viene imposto un valore nullo per lo stesso puntatore.

```
Studente * Rubrica::ricerca(char * chiave)   // ricerca esaustiva
{
int i;
for (i=0;i<num;i++)
  if (strcmp(vs[i].get_cognome(), chiave)==0)
    return ( &(vs[i]) );

return(NULL);
}
```

L'operazione di ricerca realizzata si svolge in un numero finito di passi elementari. Tali passi elementari comprendono il confronto della chiave con quella contenuta nel record di indice i, l'incremento dell'indice `i`. Il numero di confronti eseguiti per ogni ricerca dipende dalla posizione del record da cercare. Nel caso migliore (tempo di esecuzione minore) il ciclo `for` viene eseguito solo una volta,

l'oggetto cercato e' il primo, si opera un solo confronto con la chiave. Il caso peggiore si ha quando l'elemento da cercare e' nell'ultima posizione del vettore. In tal caso si operano `num` confronti. Ogni confronto della chiave si basa su `S` confronti a livello di carattere, si operano pertanto `S*num` confronti a livello di carattere (se le chiavi hanno tutte lunghezza pari a `S` caratteri).

15.5 Cancellazione in un vettore ordinato: la classe Rubrica

L'operazione di cancellazione implica una riorganizzazione del vettore. Se l'elemento da cercare e' l'ultimo non vi sono problemi si può semplicemente cancellarlo riducendo di una unita' il numero degli elementi del vettore. Per essere pignoli si può copiare sull'oggetto cancellato in ultima posizione un oggetto vuoto.

```
void Rubrica::cancella(int pos) // cancellazione
{ int i;
Studente s; // uno studente vuoto
if (pos!=DIMRUB-1) for (i=pos;i<num;i++) vs[i]=vs[i+1];
else vs[pos]=s; // considerazione dell'ultimo elemento

num--;   // aggiorno il numero degli elementi
}
```

Può essere necessario poter effettuare la cancellazione anche a partire dal puntatore. Questa operazione e' specialmente utile perché il metodo di ricerca produce in uscita il puntatore all'elemento trovato. Dal punto di vista computazionale e' un complicazione poiché dato il puntatore e' necessario scorrere il vettore per trovare l'elemento da cancellare.

```
int Rubrica::cancella(Studente * pastud) // cancellazione
{
int i, pos=DIMRUB+1;

for (i=0;i<num;i++) if ( pastud==&(vs[i]) ) pos=i;
if (pos==DIMRUB+1) return (1);

cancella(pos);  // cancellazione
return(0);
}
```

Tale operazione può anche essere evitata utilizzando la matematica dei puntatori come segue. Per esempio con: `pos=(int)(pastud-&vs[0]);` al posto del ciclo `for`. La conversione di tipo può non essere necessaria se il compilatore ha un basso livello di attenzione agli errori marginali, altrimenti e' necessaria.

```
int Rubrica::cancella(Studente * pastud) // cancellazione
{ int pos;
pos=(int)(pastud-&vs[0]);
cancella(pos);  // cancellazione
return(0);
```

```
}
```

Questa soluzione può essere adottata anche nei successivi metodi di modifica.

15.6 Modifica di un record: la classe Rubrica

La modifica di un record non e' un problema se non viene modificato il campo chiave. La modifica può essere vista come una stampa dei singoli elementi dell'oggetto e il successivo reinserimento dei valori corretti. In accordo al paradigma object-oriented se si desidera effettuare la modica delle informazioni dello studente e quindi anche dell'indirizzo tale operazione deve essere delegata alla classe Indirizzo. Pertanto un metodo di modifica può avere la seguente struttura dove una volta identificato l'elemento da modificare viene delegata la modifica ad un metodo della classe Studente.

```
int Rubrica::modifica(Studente * pastud) // cancellazione
{
int i, pos=DIMRUB+1;
for (i=0;i<num;i++) if ( pastud==&(vs[i]) ) pos=i;
if (pos==DIMRUB+1) return (1);

vs[pos].modifica();
return(0);
}
```

Il metodo `modifica()` per la classe studente riportato di seguito e' una versione modificata rispetto al metodo presentato in precedenza. La differenza consiste nell'aver commentato le istruzioni relative alla modifica del campo chiave.

```
void Studente::modifica(void)
{
printf ("Nome = %s : ", nome); scanf("%s", nome);
// questo non puo' essere cambiato mantenendo la posizione
// printf ("cognome = %s : ", cognome); scanf("%s", cognome);
printf ("tel = %s : ", tel); scanf("%s", tel);
printf ("eta' = %d : ", eta); scanf("%d", &eta);
indir.modifica();
}
```

Questo impedisce di modificare il campo chiave e quindi di rendere non valido l'ordinamento degli elementi della Rubrica a causa della modifica di un singolo oggetto.
Se tale operazioni viene effettuata la presenza di un vettore non piu' ordinato invalida le successive operazioni di ricerca, cancellazione, inserzione, etc.

15.7 Modifica del campo chiave: la classe Rubrica

Quando si ha una modifica del campo chiave, in generale l'oggetto non risulta più nella posizione corretta rispetto agli altri elementi in base al criterio di ordinamento utilizzato per la struttura dati.

Se l'operazione di modifica del cambio chiave e' necessaria deve essere possibile vedere se vi e' stato un cambio di chiave. In tale caso, si può

1. salvare l'oggetto,
2. cancellarlo dalla struttura e
3. reinserirlo nella giusta posizione.

Di seguito viene riportato il metodo modifica per la classe Rubrica che permette anche la modifica del campo chiave. Tale metodo deve utilizzare il metodo `Studente::modifica()` nella versione originale senza i commenti.

```
int Rubrica::modifica(Studente * pastud) // cancellazione
{
int i, pos=DIMRUB+1, ret=0;
char nometmp[DIMSTR];
Studente std;

for (i=0;i<num;i++) // ricerca dello studente da modificare
  if ( pastud==&(vs[i]) )
    pos=i;

if (pos==DIMRUB+1) return (1);

strcpy(nometmp, vs[pos].get_cognome() ); // salvo il cognome

vs[pos].modifica(); // modifico

// verifico se vi e' stata una modifica del campo chiave
if ( strcmp(nometmp, vs[pos].get_cognome())!=0 ) // cambio chiave
  {
  std=vs[pos];   // salvataggio dello studente
  cancella(pos);  // cancellazione
  ret=inserimento(std);  // reinserimento nella nuova posizione
  if (ret!=0)
    return(1); // inserimento senza successo
  }

return(0); // modifica con successo
}
```

15.8 Classe Rubrica: programma di collaudo

Di seguito e' riportato il programma principale che permette di collaudare ed utilizzare la `Rubrica` presentata in precedenza con le classi `Studente` e `Indirizzo`. Si noti la realizzazione di un piccolo menu di scelte effettuato per mezzo di alcune chiamate alla funzione `printf()` e della

funzione `getch()` per acquisire la scelta. I vari casi, contengono anche le istruzioni per gestire i valori di ritorno di vari metodi della classe `Rubrica`. Si noti come le operazioni di cancellazione e modifica iniziano con una ricerca per identificare l'oggetto ove tali operazioni devono essere effettuate.

```
#define DIMSTR 100

void main(void)
{
char c, buf[DIMSTR];
Rubrica lamia;
Studente stmp, *pstmp; // puntatore a uno studente temporaneo
int ret, flg=0;

pstmp = new Studente(); // allocazione di uno studente

do { // visualizzazione del menu
   printf("\nRubrica, operazioni: \n\n");
   printf("a) inserisci \n");
   printf("b) stampa \n");
   printf("c) ricerca \n");
   printf("d) cancella \n");
   printf("e) modifica \n");
   printf("q) esci \n\n");
   printf("Premi un tasto: ");

   c=getch(); // acquisizione della scelta
   printf("\n");

   // azioni associate alle scelte
   switch(c)
      { case'a': case 'A': // inserimento
         stmp.getdati();
         ret=lamia.inserimento(stmp);
         if (ret!=0)
            printf("Rubrica completa ! Operazione impossibile\n");
         break;

      case 'b':  case 'B': // stampa
         lamia.stampa();
         break;

      case 'c':  case 'C':  // ricerca
         printf("Studente da cercare, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricerca(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         pstmp->stampa();
         break;
```

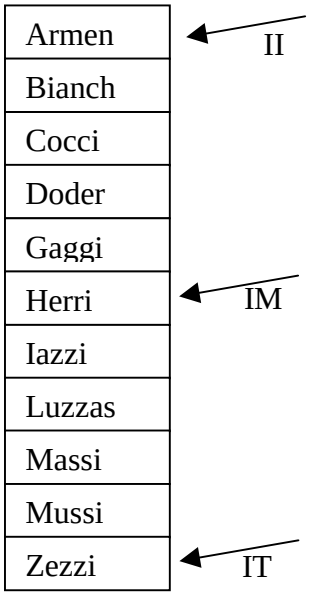
```
      case 'd':  case 'D': // cancellazione
         printf("Studente da cancellare, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricerca(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         ret=lamia.cancella(pstmp);
         if (ret!=0) printf("problemi di Cancellazione\n");
         break;

      case 'e':  case 'E': // modifica
         printf("Studente da modificare, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricerca(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         ret=lamia.modifica(pstmp);
         if (ret!=0) printf("problemi di Modifica\n");
         break;

      case 'q':  case 'Q': // uscita
         flg=1;
         break; // uscita

      default: printf("Operazione non consentita \n"); break;
      }
   if (flg==0)
      { printf("Premi un tasto per un'altra operazione\n");
      getch();
      }
   } while (flg!=1);
}
```

Ogni scelta del menu e' stata realizzata considerando la pressione di lettere maiuscole e minuscole. Questo conferisce maggiore robustezza al programma, coprendo eventuali errori dell'utente. E' stato inoltre inserito nel caso di default un messaggio per avvisare l'utente quando questo effettua delle scelte non contemplate dal menu stesso.

La funzione `getch()` e' utilizzata nel menu per acq1uisire il carattere di scelta e in fondo al programma per attendere che l'utente legga un messaggio di aiuto.

Si noti il metodo utilizzato per uscire dal programma. Quando viene permuto il tasto `'q'` o `'Q'` si assegna il valore `1` ad un flag, `flg`, che e' inizialmente posto a `0` ad ogni avvio del programma. Il ciclo `do-while` che permette di effettuare altre operazioni ha come condizione di uscita proprio il valore di del flag `flg`; se questo assume il valore 1 allora la condizione non e' soddisfatta e si esce dal ciclo portando il programma a chiudersi.

15.9 Ricerca dicotomica su vettore ordinato



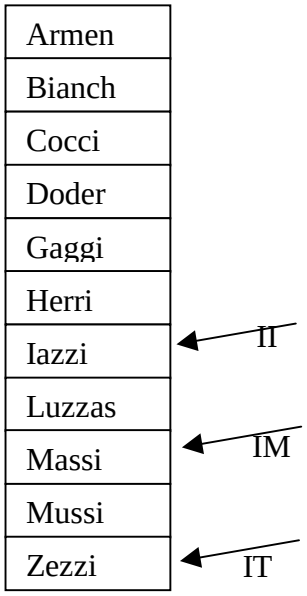
Il metodo di ricerca esaustiva esposto in precedenza non e' molto efficiente poiché la ricerca su un vettore ordinato di N elementi viene effettuata sulla base di N confronti (nel caso di ricerca senza successo). Il metodo di ricerca dicotomica (detta anche ricerca binaria o logaritmica) permette di ridurre notevolmente il numero dei confronti e quindi i tempi di ricerca. Questo metodo si applica solo a insiemi ordinati e si basa su un principio semplice: l'insieme viene diviso in due parti e si prosegue la ricerca sulla parte che contiene la chiave cercata. Nell'esempio riportato a fianco si ha un vettore ordinato di oggetti dei quali e' riportato solo il campo chiave. Se si ricerca l'oggetto con chiave Mussi *dividendo il vettore in due parti, da II a IM e da IM a IT e' chiaro che considerando l'oggetto identificato da IM, l'oggetto con chiave* Mussi *sarà nella prima o nella seconda meta' oppure sarà proprio quello identificato da IM.*

Questo può essere facilmente verificato confrontando il campo chiave dell'oggetto identificato da IM con la chiave. Se la chiave risulta piu' pesante di quella dell'oggetto allora sarà nella seconda parte, etc. II sta per indice iniziale, IM per indice mediano e IT per indice terminale.
Questo procedimento di divisione ha ridotto lo spazio di ricerca di ½, e quindi anche il numero di confronti da fare.

Dato II ed IF generici in un vettore IM si può calcolare come il valor medio arrotondato all'intero inferiore:

$$IM = \left\lfloor \frac{(II + IT)}{2} \right\rfloor$$

Questo procedimento può essere applicato nuovamente alla parte che contiene l'oggetto cercato semplicemente aggiornando il valore di II ed IT in modo opportuno. Il procedimento converge quando i IT>= II oppure quando si trova l'elemento cercato.
Il codice relativo a questo algoritmo e' riportato di seguito.
Nel codice sono presenti anche due istruzioni commentate che possono essere utilizzate per verificare il funzionamento dell'algoritmo in fase di collaudo.



```
Studente * Rubrica::ricercaDT(char * chiave)
{
int ii=0, it=num, im, ret;

if (num==0) return NULL; // vettore vuoto

do { im=(it+ii)/2;
//  printf("ii %d, it %d, im %d\n", ii, it, im);

  ret=strcmp(vs[im].get_cognome(), chiave);
  if ( ret>0 )  // di sopra
     { it=im-1; }
  else if ( ret<0 ) // di sotto
     { ii=im+1; }
  else return ( &(vs[im]) );
  } while (it>=ii);

// printf("esco con ii %d, it %d, im %d\n", ii, it, im);

return(NULL);
}
```

Questo metodo di ricerca può essere inserito nel programma principale presentato in precedenza al posto della ricerca esaustiva.

Traccia della ricerca con successo dell'oggetto con chiave *Mussi*

```
ii 0, it 11, im 5
ii 6, it 11, im 8
ii 9, it 11, im 10
ii 9, it 9, im 9
```

Traccia della ricerca con senza successo dell'oggetto con chiave *Borgi*

```
ii 0, it 11, im 5
ii 0, it 4, im 2
ii 0, it 1, im 0
ii 1, it 1, im 1
esco con ii 2, it 1, im 1
Studente non trovato
```

Il procedimento esposto procede a dimezzare il vettore ad ogni iterazione analizzando sono un sottovettore. La prima volta il vettore viene ridotto a ½ la seconda ad ¼ , etc. fino a che (nel caso peggiore di ricerca senza successo) il sottovettore non si riduce ad un solo elemento. Al passo `i`-esimo il vettore sarà composto da $\lfloor N / 2^{i-1} \rfloor$ elementi.

L'algoritmo si arresta quando il sottovettore ha lunghezza unitaria:

$$\left\lfloor \frac{N}{2^{i-1}} \right\rfloor = 1$$

dalla quale si ricava che si arresta per $i = \lfloor Log_2 N \rfloor + 1$

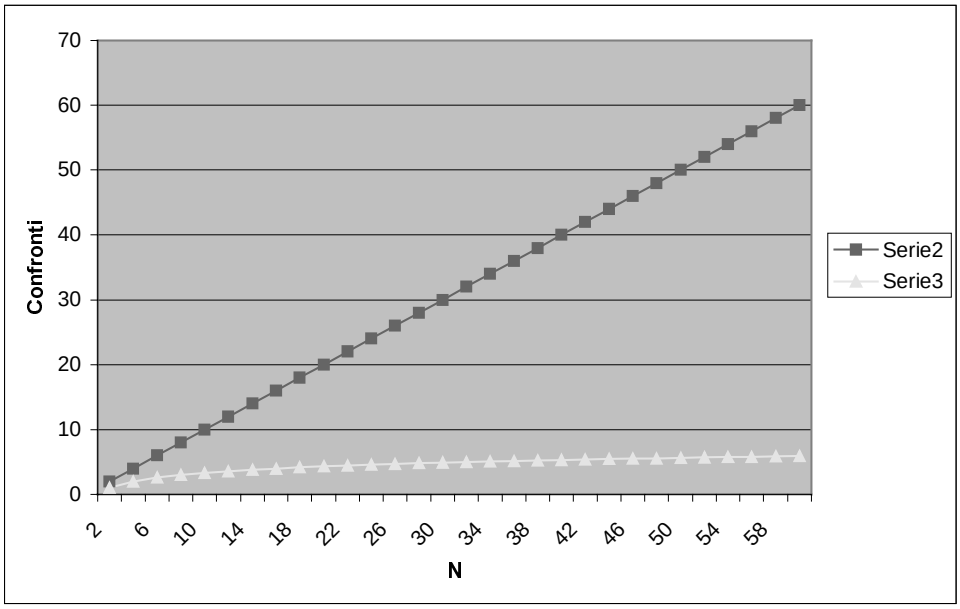
Si può pertanto affermare che la ricerca dicotomica ha una complessità asintotica pari ad un O(Log(N)).

15.10 Ricerca Binaria, Confronto fra O(N) e O(Log N)

Sulla base della complessità l'algoritmo di ricerca dicotomica e' sicuramente piu' veloce rispetto a quello esaustivo. Nella tabella sono riportati dei valori che danno un'idea del numero dei confronti, i, che sono necessari quando si ha a che fare con vettori di N elementi anche di dimensioni considerevoli. Si noti che con vettori dell'ordine di 10^{60} elementi sono necessari solo 200 confronti.

$N = 2^I - 1$	**Confronti, i**
1	1
1023	10
1,13E+15	50
1,27E+30	100
1,61E+60	200

Nel grafico a fianco l'andamento di del numero dei confronti in funzione del numero degli elementi per O(N) (grafico in rosa, serie 2) e O(Log N) (grafico in giallo, serie 3).

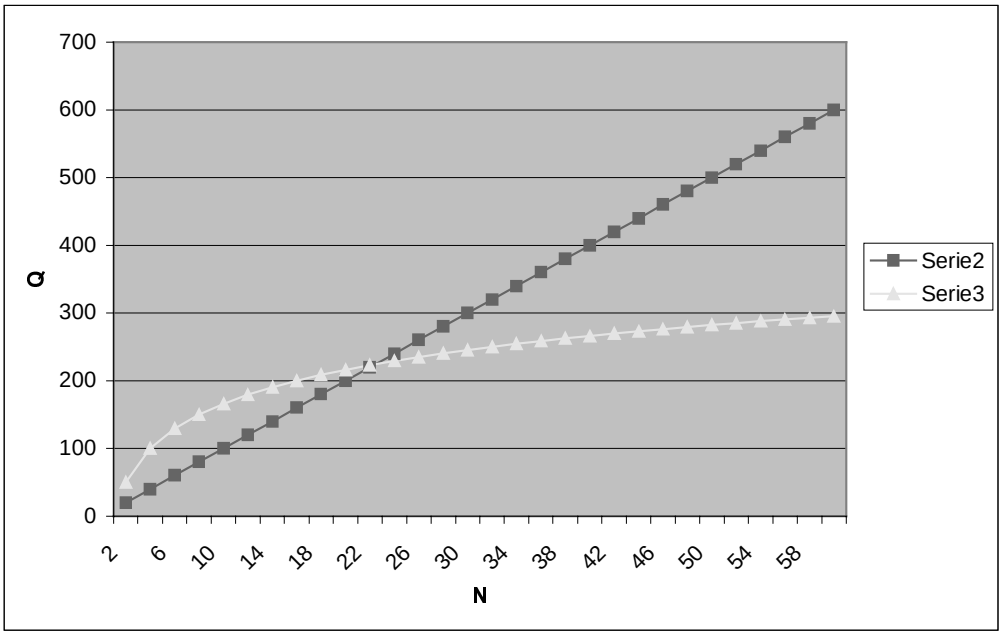


Dal grafico sembra evidente che non vi sono valori di N per i quali LogN risulta peggiore. Per ogni valore di N, algoritmi con O(N) necessitano di un maggior numero di confronti.
In effetti questa condizione non e' sempre vera quando si vanno a considerare le complessità di dettaglio e non quelle asintotiche.

15.11 Confronto fra (k N) e (H Log N)

E' interessante fare un'analisi dell'andamento di due funzionali di costo: K*N e H Log N, (con Log in base 2). Questi non rappresentano piu' il numero di confronti ma il numero delle operazioni elementari Q effettuate dall'algoritmo.

Dal punto di vista asintotico sicuramente Log N e' da preferire ma per valori piccoli di N con valori diversi di K ed H LogN presenta un comportamento che può essere peggiore di quello di N.



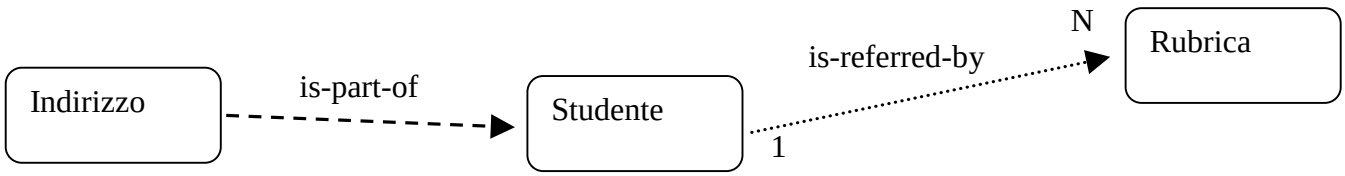
Per esempio in figura e' stato confrontato 10 N (in rosa, serie 2) con 50 LogN (in giallo, serie 3). Da questo grafico si evince che in questo caso e' preferibile utilizzare un algoritmo che ha 10 N se N minore di circa 22 e utilizzare l'algoritmo che ha 50LogN solo quando N e' grande. Il confronto su base asintotica, cioè basato sulla complessità asintotica ha senso solo per N che tende a infinito o quanto meno per N "grande". La misura di questo "grande" dipende dai fattori scale del problema e dalle costanti, per esempio K ed H, rispettivamente 10 e 50, in questo esempio.

15.12 Vettore ordinato di puntatori: la classe RubricaVP

Nel caso in cui si utilizzi un vettore di puntatori invece che un vettore statico si devono fare svariate modifiche ai metodi della classe Rubrica.

```
class RubricaVP
{
    int num;  // numero degli elementi
    Studente *pvs[DIMRUB];
public:
     ……….
};
```

Utilizzando questo tipo di soluzione invece che il vettore di Studenti la sostanza non cambia ma si ha un miglioramento dello sfruttamento della memoria come e' stato mostrato, ed inoltre si ha un miglioramento delle prestazioni poiché durante le fasi di scorrimento dei record (nell'inserimento) per fare il posto ad un nuovo elemento e nella fase di ricompattazione (dopo la cancellazione) non si devono effettuare le copie dei singoli oggetti ma solo le copie dei singoli puntatori. Questa operazione può essere molto dispendiosa quando si ha a che fare con strutture dati/oggetti di grandi dimensioni. Infatti, se si lavora con puntatori e' sufficiente copiare il valore di un puntatore.



15.13 Concatenazione di vettori ordinati

15.14 Complessità nella gestione di Vettori ordinati

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di una certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni.

- Per inserimento si intende l'operazione di inserzione senza considerare la ricerca ma considerando eventuali operazioni di riorganizzazione.
- Per cancellazione si intende la sola operazione di eliminazione dell'oggetto senza considerare la ricerca ma considerando le eventuali operazioni di riorganizzazione.
- Per modifica si intende la sola operazione di modifica dell'oggetto senza considerare la ricerca e senza considerare il caso in cui si ha un cambio chiave e quindi si deve cancellare e reinserire.

Nella seguente tabella sono riportate le complessità asintotica, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su un vettore ordinato di N oggetti (per esempio un vettore di

studenti nella classe `Rubrica` o un vettore di puntatori ad oggetti nella classe `RubricaVP`). Fra queste operazioni l'unica globale e' quella di stampa, le altre sono operazioni sostanzialmente sul singolo oggetto.

La tabella riassuntiva:

Vettore ordinato di N elementi				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(N)	N/2	1	N
Cancellazione	O(N)	N/2	1	N
Ricerca	O(N)	N/2	1	N
Ricerca dicotomica	O(LogN)	LogN/k	1	O(LogN)
Modifica	O(1)	1	1	1
Stampa	O(N)	N	N	N

16 Modulo: la ricorsione

Descrizione del modulo da scrivere

16.1 Le funzioni ricorsive

Per funzione ricorsiva o algoritmo ricorsivo si intende un procedimento/algoritmo definito in termini di se stesso. Molte funzioni matematiche semplici possono essere definite in modo ricorsivo. Per esempio anche il prodotto di due numeri può essere dato in termini ricorsivi. In seguito viene riportata la definizione non ricorsiva e quella ricorsiva.

```
P(q,r) = q * r

se (q>0) allora P(q,r) = r + P(q-1,r)
altrimenti      P(q,r) = r
```

In questo primo esempio si nota come la definizione ricorsiva sia composta da due parti:

1) l'iterazione elementare. In questo caso `P(q,r) = r + P(q-1,r)`
2) il criterio di arresto, nel caso soprastante `P(q,r) = r`

Il criterio di arresto permette di contenere il procedimento ricorsivo che altrimenti potrebbe in teoria procedere in modo indefinito.

Per ogni algoritmo ricorsivo esiste una versione non ricorsiva, tipicamente iterativa. In alcuni casi la versione ricorsiva e' preferibile mentre in altri e' senza dubbio piu' opportuno utilizzare la funzione iterativa.

16.2 La ricorsione: il fattoriale

Un numero fattoriale viene rappresentato dal numero stesso con un punto esclamativo.
La definizione piu' naturale per il calcolo del fattoriale e' quella basata sul principio di induzione:

```
se n<=1 allora  n! = 1
altrimenti n! = n (n-1)!
```

In questo modo, il fattoriale di un numero `n` viene definito specificando rispettivamente il passo base per `n <= 1` e il passo di induzione per gli altri valori di `n`. E' opportuno osservare che la definizione del passo base permette di definire il fattoriale anche per `n=0` come `0!=1`.

L'opportunità trae origine dalla necessita' di mantenere la congruenza per alcune equazioni del calcolo combinatorio. Ad esempio le combinazioni di `n` elementi a classe `n` sono manifestamente 1. Poiché il numero di tali combinazioni si ottiene mediante il coefficiente del binomiale:

$$\binom{n}{n} = \frac{n!}{n!(n-n)!}$$

e' necessario definire `0!=1` per poter calcolare il coefficiente binomiale in modo congruente. Il passo base proposto estende la definizione della funzione anche ai numeri negativi, in modo da considerare i valori che possono essere assunti dal tipo di dato ``intero con segno" normalmente disponibile nei linguaggi di programmazione.

La possibilità di usare la ricorsione nei linguaggi di programmazione permette la traduzione diretta della definizione in un programma. La funzione Fattoriale traduce in modo diretto la definizione algebrica data, separando passo base e passo di induzione. Quest'ultimo e' reso possibile dal costrutto linguistico della ricorsione che e' utilizzabile in Pascal (all'interno di una funzione si può richiamare la funzione stessa). Se `n <=1` la funzione restituisce subito il risultato (passo base). Per `n>1` la funzione richiama se stessa fino a ridurre `n=1`. Quando questa Condizione viene verificata, inizia il calcolo a ritroso di Fattoriale(2), Fattoriale(3), .. fino a Fattoriale(n) utilizzando il passo di induzione.

```
long int fattoriale(int n)
{
if(n<=1)
  return 1; /* passo base */
else
  return n*fattoriale(n-1); /* passo di induzione */
}
```

Questo e' il programma di collaudo:

```
void main(void)
{
int numero;

printf("\nInserisci il numero ");
scanf("%d", &numero);

printf("Il fattoriale di %d e' %ld\n",numero,fattoriale(numero));
}
```

Nella codifica e' stato utilizzato un parametro di ritorno di tipo `long int` per poter rappresentare il fattoriale di numeri significativi. La funzione fattoriale poteva essere realizzata anche in modo iterativo considerando una definizione basata sulla prodottoria.

$$n! = \prod_{i=1}^{n} i$$

```
long int fattorialeIT(int n)
{
long i,p=1;
for (i=1;i<=n;i++) p=p*i;
return p;
}
```

16.3 La ricorsione: la ricerca dicotomica

La ricorsione non viene utilizzata solo per la definizione di algoritmi prettamente matematici ma può essere utilizzata anche per altri tipi di algoritmi. In informatica, ed in particolare nella navigazione delle strutture dati, la ricorsione viene utilizzata di frequente. Anche l'algoritmo di ricerca dicotomica esposto e discusso in precedenza può essere dato in forma ricorsiva. A questo fine e' necessario identificare il passo elementare.

La ricerca in un sottovettore che parte da II a IF si effettua dividendo il sottovettore in due parti e sulla base di un confronto vedere su quale parte deve continuare la ricerca (riapplicando questo stesso procedimento) oppure se l'elemento e' stato trovato.

Il criterio di arresto si basa su quello visto in precedenza, o *l'elemento viene trovato oppure i due indici II e IF vanno ad assumere lo stesso valore.*

La codifica dell'algoritmo in forma ricorsiva e' semplice. La procedura ricorsiva ha un diverso numero e tipo di parametri rispetto alle procedure di ricerca utilizzate in precedenza pertanto sono state realizzate due procedure di ricerca.

La prima serve per innescare il procedimento ricorsivo e fa da interfaccia verso il programma principale che in questo modo può lavorare con la stessa modalità con tutti e tre i metodi di ricerca.

```
Studente * Rubrica::ricercaDTR(char * chiave)
{
int ii=0, it=num, ret;
if (num==0) return NULL; // vettore vuoto

ret=ricercaDTR(ii,it, chiave);
if (ret<0) return (NULL);
return ( &(vs[ret]) );
}
```

Da tale procedura di ricerca si chiama la procedura riportata in seguito che realizza la ricerca dicotomica in forma ricorsiva. Si noti che la procedura e' definita in termini di se stessa.

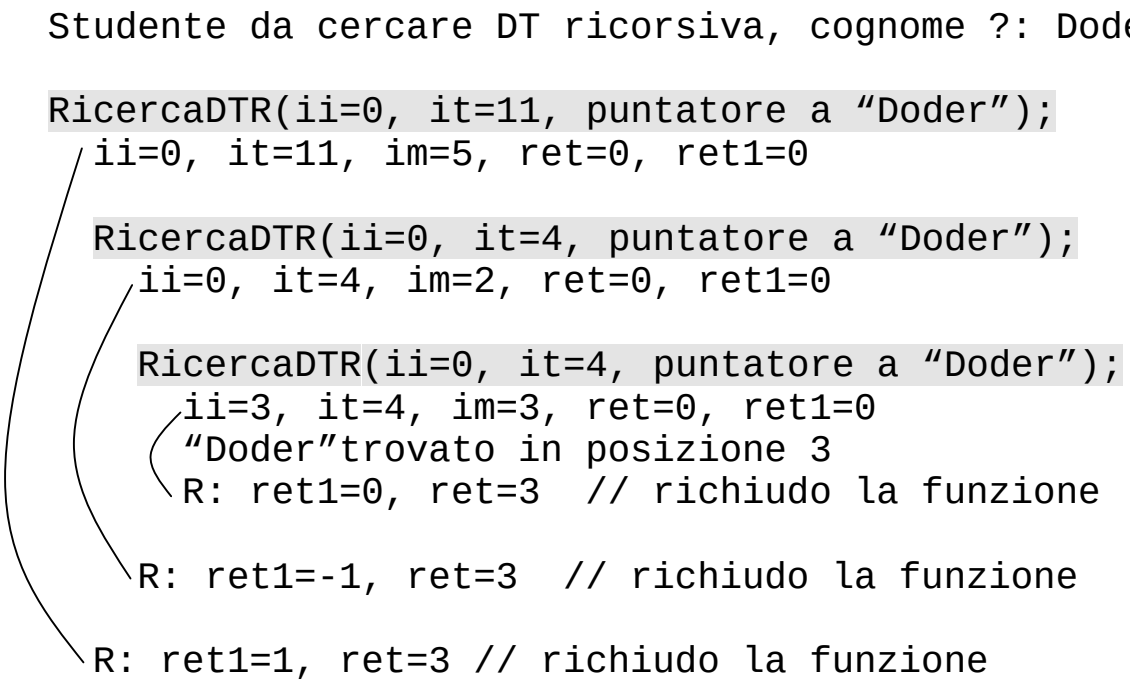
Dopo aver verificato se vi sono i presupposti per effettuare una divisione si calcola IM. In seguito si confronta la chiave con il valore di chiave dell'oggetto indicato da IM e si procede a scegliere la strada: applicare la stessa procedura al sottovettore superiore o inferiore. Questa procedura ritorna come valore l'indice dell'oggetto trovato oppure un valore negativo in caso di ricerca senza successo.

```
int Rubrica::ricercaDTR(int ii, int it, char *chiave)
{
int ret1, ret, im;

if (it<ii) return (-1);
im=(it+ii)/2;

// printf("R: ii %d. it %d, im %d\n", ii, it, im);  // traccia
ret1=strcmp(vs[im].get_cognome(), chiave);

if ( ret1>0 )      // nella parte superiore
   ret=ricercaDTR(ii, im-1, chiave);
else if ( ret1<0 ) // nella parte inferiore
   ret=ricercaDTR(im+1, it, chiave);
else
   ret=im;        // oggetto trovato

return(ret);
}
```

Se si utilizza lo stesso vettore utilizzato in precedenza si ottiene la stessa traccia di esecuzione ed evoluzione degli indici mostrata in precedenza.

16.4 La ricorsione e lo stack

Lo *stack* viene pesantemente sfruttato dai programmi ricorsivi. Questo e' dovuto al fatto che in tali tipi di soluzioni si arriva al criterio di arresto dopo avere sviluppato un numero considerevole di chiamate annidate a sottoprogrammi.
Per esempio se si considera il vettore di collaudo considerato in precedenza e si effettua la ricerca dell'oggetto con chiave "Doder", si ha la seguente situazione. La prima chiamata viene effettuata con i valori di `ii=0` e `it=11` e nello *stack* viene caricato il valore di tali parametri (`ii, it,` puntatore a chiave) piu' le variabili locali della procedura: `im, ret, ret1` ed il valore di ritorno della procedura per un totale di `6 int` (`6*2` pari a `12` byte in MS DOS) e un puntatore a `char` (4 byte). 16 byte in tutto.



Ad ogni divisione del vettore viene chiamata un’altra istanza della stessa procedura con diversi valori dei parametri e questi vanno ad inserirsi nello *stack* fino a che non si arriva alla chiamata che provoca un’uscita dalla procedura stessa. In questo caso si ha la chiamata al terzo livello nella quale si vede che `ret1` assume il valore 0 (chiave trovata). Questo provoca la chiusura del sottoprogramma e dopo di questo quello del sottoprogramma chiamante, fino alla chiamata iniziale. Questo significa che se si hanno H livelli ed ad ogni livello si allocano nello *stack* M celle di memoria all’ultimo livello lo *stack* conterrà H*M celle di memoria con il rischio di andare in *stack overflow*. Tale problema viene riscontrato solo durante l’esecuzione del programma e provoca spesso l’arresto drastico del programma stesso.

Tale problema può essere riscontrato facilmente quando si hanno delle procedure ricorsive che hanno come parametri passati per valore strutture dati o oggetti di dimensioni considerevoli.

Di seguito e’ riportata la traccia dell’esecuzione dell’algoritmo di ricerca senza successo. In tale case i livelli di chiamata sono stati 4. Ogni livello e’ indentato di due caratteri verso destra.

```
Studente da cercare DT ricorsiva, cognome ?: Mezza
  ii=0, it=11, im=5, ret=0, ret1=0
    ii=6, it=11, im=8, ret=0, ret1=0
      ii=9, it=11, im=10, ret=0, ret1=0
        ii=9, it=9, im=9, ret=0, ret1=0 // criterio di arresto valido
        R: ret1=1, ret=-1
      R: ret1=1, ret=-1
    R: ret1=-1, ret=-1
  R: ret1=-1, ret=-1
Studente non trovato
```

16.5 Criteri di scelta per la ricorsione

In generale per ogni soluzione ricorsiva si può realizzare una soluzione iterativa e viceversa. In alcuni casi e' possibile avere algoritmi in forma chiusa che permettono di arrivare alla soluzione per via matematica senza l'utilizzo di iterazioni o ricorsioni.

Spesso le soluzioni ricorsive sono piu' eleganti e semplici, presentano un minor numero di istruzioni. Pero' non sempre sono le piu' semplici da comprendere e realizzare, ne tantomeno sono le piu' veloci da eseguire. Spesso impegnano troppo lo *stack*.

Sulla base di questi fattori si possono dare dei criteri per poter scegliere la migliore fra due soluzioni di uno stesso problema in forma iterativa o ricorsiva.

In generale, la soluzione iterativa deve essere preferita quando:

- si conosce in modo esatto il numero delle iterazioni
- l'uso dello stack non e' funzionale al programma, cioè nello stack vanno a finire molte copie delle stesse variabili poiché queste figurano fra i parametri della procedura. Una parziale soluzione e' l'uso di puntatori.
- Esiste una soluzione iterativa o in forma chiusa che permette di ottenere buone prestazioni senza un eccessivo uso della memoria di base o dello stack.

La soluzione in forma ricorsiva e' da preferire quando:

- Non si conosce a priori il numero delle iterazioni
- E' di gran lunga piu' semplice rispetto a quella iterativa, spesso perche' lo stack mantiene traccia del percorso e non e' necessario replicare tale storia in variabili temporanee.
- Il numero dei livelli di chiamata e' limitato e non dipendente da parametri che raggiungono valori elevati. Per esempio e' sconsigliabile l'uso della ricorsione in algoritmi di complessità N che provocano N chiamate annidate.

Per queste ragioni la ricorsione non e' giustificata per il calcolo del fattoriale che può essere stimato semplicemente in forma iterativa.

La realizzazione in forma ricorsiva dell'algoritmo di ricerca dicotomica e' piu' giustificata. L'ammontare delle informazioni passate come parametri e' limitato, non si conosce il numero delle divisioni a priori, il numero di livelli di chiamata va con il Logaritmo del numero degli elementi del vettore.

Anche l'algoritmo di ricerca esaustiva può essere realizzato in forma ricorsiva, ma in tale caso l'adozione di tale soluzione non e' giustificato e quindi deve essere caldamente evitato. Lo stack verrebbe ad essere impegnato con complessità pari ad un O(N).

16.6 Ricorsione Indiretta

17 Modulo: I File.

I file sono dei contenitori di dati per la memoria di massa. In tali contenitori possono essere immagazzinate informazioni di varia natura. Le operazioni che si possono compiere sono principalmente di scrittura e lettura dati.

La natura dei file può essere ASCII cioè testuale oppure binaria. I file ASCII sono facilmente riconoscibili poiché il loro contenuto può essere visionato semplicemente utilizzando il comando *type* oppure per mezzo dell'editor stesso con il quale si scrivono i programmi. Per file ``visionabile'' si intende che questo e' composto di caratteri comprensibili, come le lettere minuscole e maiuscole, i segni di punteggiatura, i numeri, ma non i simboli strani e caratteri ASCII che identificano i simboli di controllo. File ASCII vengono detti anche in chiaro poiché e' possibile leggere il loro contenuto come si legge un documento, i dati contenuti non sono codificati.

Anche i file binari possono essere visualizzati con il comando *type* e con un editor ma il loro contenuto risulta essere composto da caratteri non solamente alfanumerici e quindi di non facile comprensione.

Pertanto quello che contraddistingue i file visionabili e' il fatto che contengono prevalentemente testo comprensibile. L'uso delle procedure per leggere e scrivere su disco permette di rendere i dati e/o risultati permanenti e quindi rianalizzabili, riutilizzabili, etc.

17.1 I nomi dei file

Ogni file deve avere un nome univoco, che permette di distinguerlo da altri nello stesso *contenitore* di file. Sono nomi di file:

```
pippo.txt
programma2.c
log.cpp
relazione.doc
```

La lunghezza del nome e i caratteri che possono essere utilizzati per scrivere tale nome dipendono dal sistema operativo utilizzato. In MSDOS i nomi erano limitati a 8+3 caratteri mentre in Windows 98 o NT tale limitazione e' stata soppressa. E' comunque più semplice adottare per i nomi dei programmi una limitazione 8+3. 8 e' il numero dei caratteri del nome, 3 il numero dei caratteri che determina l'estensione del file. L'estensione del file e' un modo immediato per dichiarare il tipo di file. Per esempio i file testuali hanno tipicamente estensione TXT, quelli prodotti da molti editor sofisticati per la produzione di documenti testuali hanno estensione DOC. I file eseguibili hanno estensione EXE, COM o BAT.

Nei sistemi operativi ad ogni file vengono associate diverse informazioni: data della creazione, lunghezza, tipo di file come archivio, lettura, scrittura, etc. Questi dettagli vengono discussi in seguito.
I file possono essere contenuti in cartelle dette Directory. Una memoria di massa puo' avere molte directory e queste possono essere annidate fra di loro. In ogni directory non possono esistere file con lo stesso nome.

17.2 Le operazioni di base dei file

Le operazioni di base sui file sono:

- Apertura del file: richiesta da parte del programma al sistema operativo di aprire un file su disco: questo può essere già presente o meno. Se non e' presente il file può essere creato o meno. Una volta che il file e' aperto questo viene controllato solo dall'applicazione che lo ha aperto. In genere, altre applicazioni che provano ad utilizzare lo stesso file non verranno soddisfatte.
- Inserimento di dati nel file: inserimento di dati in posizioni specifiche, inserimento alla fine del file. Operazione possibile solo se il file e' aperto.
- Lettura di dati da file, lettura in un punto specifico, lettura a partire dall'inizio del file in modo sequenziale. Operazione possibile solo se il file e' aperto.
- Chiusura del file, rilascio del file da parte dell'applicazione. Operazione possibile solo se il file e' aperto.

In alcuni casi le operazioni di inserimento/scrittura e lettura/estrazione di dati dipendono dal tipo di file e da come questo e' stato aperto.
Tramite degli strumenti del sistema operativo e' possibile impostare le proprietà di un file in modo che questo non possa essere manipolato in modo improprio. Per esempio e' possibile impedirne la cancellazione, la scrittura ed anche la lettura.

17.3 Apertura dei File, la funzione `fopen()`

L'operazione di apertura dei file pregiudica le funzionalità del file stesso. In C/C++, si possono aprire dei file per mezzo della funzione di libreria `fopen().`

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

Tale funzione di libreria presenta due parametri e ritorna un identificato del FILE da utilizzare per le successive operazioni. Tale identificativo viene chiamato *Handle*, maniglia.
Per esempio con la seguente serie di istruzioni viene aperto un file di nome prova1.txt e viene scritto in tale file la stringa "questo e' un file":

```
void main(void)
{
FILE *gt;
gt=fopen("prova1.txt","w+");
if (gt==NULL) printf("Impossibile aprire il file\n");
else
  fprintf(gt,"questo e' un file\n");
fclose(gt);
}
```

Si noti nel precedente gruppo di istruzioni che l'apertura del file con `fopen()` produce un valore di ritorno che e' stato assegnato alla variabile `gt` di tipo `FILE *.` Questa e' l'*handle* del file. Tale variabile viene utilizzata dalla funzione `fprintf()` per scrivere nel file. Quando l'operazione di apertura del file non ha successo, per esempio a causa della mancanza di spazio su disco oppure per la mancanza di un file da leggere, la funzione `fopen()` ritorna un valore `NULL`. Questo spiega l'istruzione `if` per identificare tale condizione e stampare un messaggio all'utente.

La funzione `fopen()` presenta come secondo parametro una stringa. Con tale stringa viene codificata la modalità di apertura del file in accordo alla seguente tabella.

Modalita'	Descrizione
`"r"`	Apertura di un file testuale per lettura
`"w"`	Apertura di un file testuale per scrittura
`"a"`	Apertura di un file testuale per appendere alla sua fine
`"r+"`	Apertura di un file testuale per lettura e scrittura
`"w+"`	Apertura di un file testuale per scrittura, se il file esiste viene cancellato e ricreato da zero
`"a+"`	Apertura di un file testuale per scrittura alla fine, se il file non esiste viene creato

La funzione di libreria `fprintf()` e' del tutto uguale alla funzione `printf()` già esposta. L'unica differenza consiste nel primo parametro che deve essere l'*handle* del file sul quale si intende scrivere.
Anche la funzione `fclose()` deve fare riferimento all'*handle.*

17.4 Salvataggio di dati in file di testo

Si realizza in questo esempio un programma che salvi su un file di testo in caratteri ASCII i dati relativi a una funzione prodotta dal calcolatore. Per esempio una funzione del tipo: y=sin(x)/x;

Al fine di utilizzare un file da un programma e' necessario dichiarare il nome del file. In questo caso e' stato scelto `"dati.dat".`

```
void main(void)
{
int i,num;
FILE *hand;
float x,y;

hand=fopen("dati.dat","w+");   // sovrascrivo il file se esiste
num=50; fprintf(hand,"%d\n",num);
for (i=1;i<=num;i++)
   { x=i*0.3;   y=sin(i*0.3)/i*0.3;
   fprintf(hand,"%f %f\n",x,y);
   }
```

```
fclose(hand);
}
```

Per la scrittura dei dati sul file e' stata utilizzata la procedura `fprintf()`. Con l'esecuzione del programma viene prodotto il file di dati la cui struttura e' la seguente. Si noti che come prima riga e' stato memorizzato il numero di coppie di punti contenute nel file. Questo permette, in fase di rilettura, di sapere subito quanti dati devono essere letti.

```
50
0.300000 0.088656
0.600000 0.084696
....    ....
....    ....
14.700000 0.005178
15.000000 0.003902
```

Come si nota il formato utilizzato per la scrittura dei dati dal programma C non e' in forma scientifica. In C si può ottenere un'uscita in forma scientifica cambiando formato di scrittura da %f a %e. Dopo la lettura della prima riga e' noto il numero di righe di cui e' costituito il file. Pertanto, si e' utilizzata una certa convenzione per scrittura che e' stata rispettata anche in lettura.

17.5 Lettura file testuali

Per la lettura di dati da disco si utilizza la funzione `fscanf()`. Con modalità del tutto simili alla funzione `scanf()`. Il file deve essere aperto con l'uso della stringa `"r+"` per definire il tipo di file da aprire. Con tale stringa si specifica che il file viene aperto in lettura e che deve esistere. In caso contrario l'*handle* assumerà un valore NULL.

Se questa condizione viene verificata allora il programma si interrompe esegue la chiamata alla funzione `return()`, che in questo contesto farà interrompere immediatamente l'esecuzione del programma.

```
#define NUM 200   /* numero di elementi */

void main(void)
{
char nome[BUFLENG]= "dati.dat";
FILE * hand;
int i, num;
float vx[NUM], vy[NUM];

hand=fopen(nome,"r+"); // apertura del file e verifica della sua
presenza
if (hand==NULL) return(1); // file non presente

fscanf(hand,"%d",&num);
```

```
if (num>NUM) { printf("Vettore dati troppo lungo"); return(1); }

/* lettura dei dati da file */
for (i=0;i<num;i++) fscanf(hand,"%f %f",&(vx[i]),&(vy[i]) );
fclose(hand);
.....
}
```

Dopo la lettura del file si possono fare delle operazioni sui dati. E' un buon esercizio per lo studente procedere con la realizzazione di procedure per il calcolo della media del valore massimo, della varianza, etc.

17.6 Gestione di file binari o tipizzati

I file possono essere anche binari oltre che ASCII o testuali. In tal caso, i dati in essi contenuti riflettono direttamente la rappresentazione in memoria delle variabili. Per esempio, nell'esercizio precedente sono stati salvati su disco dei numeri reali utilizzando 8 caratteri alfanumerici per ogni numero; nel caso in cui tali numeri fossero stati salvati in forma binaria avrebbero occupato solo xxx byte in C, cioè un numero di byte pari al loro ingombro in memoria. Si avrebbe avuto un risparmio nell'uso dello spazio su disco, un aumento della precisione, poiché ovviamente salvare in ASCII significa salvare solo una parte dell'informazione mentre salvare in binario significa salvare la copia esatta del contenuto della memoria e quindi tutta l'informazione.

D'altra parte la scrittura in ASCII, o in ``chiaro" che dir si voglia, permette all'utente di leggere facilmente i file di dati e di esportare agevolmente i dati verso altri programmi. Questo e' dovuto al fatto che per consentire lo scambio in forma binaria fra due linguaggi e/o sistemi bisogna che questi abbiano la stessa rappresentazione dei dati, numero di byte, numero di bit di mantissa e caratteristica). In ogni caso in entrambe le rappresentazioni e' necessario conoscere la struttura del file per poterlo rileggere, nel caso dei file ASCII e' facilmente deducibile visionando il file, in quelli binari e' difficilmente comprensibile dal file. Il C e Pascal utilizzano per i numeri reali due
diverse rappresentazioni.

17.7 Esempio di salvataggio e caricamento di file binari

In questa sessione viene mostrato come e' possibile salvare su file il contenuto della rubrica organizzata con un vettore presentata in precedenza e quindi ricaricarlo da file. A questo fine viene realizzata una procedura per il salvataggio su disco di tali dati. In questo modo sarà possibile memorizzare su file e rileggere il contenuto della rubrica al momento della riaccensione del calcolatore. La rubrica rimarrà memorizzata su memoria di massa. A tal fine sarà necessario salvarla prima di uscire dal programma. Le operazioni saranno effettuate utilizzando un file binario.

Per il salvataggio e il caricamento dell'intera rubrica su disco sono state realizzate le procedure riportate in seguito. Poiché i singoli oggetti sono salvati in forma binaria e' sufficiente una sola chiamata a funzione per memorizzare un singolo oggetto. Per non tentare di aprire un file che non esiste e' stata realizzata la procedura `esiste_file()` che sarà esposta in seguito. Per il corretto caricamento di tutta l'agenda e' necessario conoscere il numero di record che sono contenuti nel file, oppure e' sufficiente

accorgersi di essere arrivati alla fine del file. Per questo scopo si può utilizzare la procedura di libreria `feof()` che assume il valore vero se si e' giunti alla fine del file, cioè allo ``end of file'', EOF.

Per la scrittura e la lettura dei record in forma binaria sono state utilizzate le procedure `fread`() e `fwrite`() anziché `fscanf`() e `fprintf`(), poiché quest'ultime sono in grado di lavorare solo su file ASCII. Per aprire il file e' sufficiente chiamare la solita funzione `fopen`() con il tipo "wb+" per avere un file binario in scrittura, eventualmente da sovrascrivere se questo e' già presente su disco; il "b" specifica che il file e' binario.
Metodo di salvataggio:

```
int Rubrica::salva(char * namefile)
{ FILE *gt;
int i;
gt=fopen(namefile,"wb+"); // apertura del file
if (gt==NULL) return(1); // se non si apre allora non si puo'procedere
for (i=0;i<num;i++)  // scrittura dei singoli studenti
  fwrite( &(vs[i]), sizeof(Studente), 1, gt);
fclose(gt); // chiusura
return(0);
}
```

Funzione per la verifica dell'esistenza o meno di un file:

```
int esiste_file(char nome[])
{ FILE *hand;
hand=fopen(nome,"r"); // tentativo di apertura
if (hand==NULL) // se null allora il file non esiste
     return(0);
fclose(hand); // altrimenti il file esiste e deve essere chiuso
return(1);
}
```

Metodo di caricamento della Rubrica:

```
int Rubrica::carica(char * namefile)
{
FILE *gt;
if (num!=0) return(2); // dati presenti
if (esiste_file(namefile)) // verifica se il file esiste
  { gt=fopen(namefile,"rb+"); // solo se esiste
  while (!feof(gt)) // finche non si arriva in fondo al file
    { fread(&(vs[num]),sizeof(Studente),1,gt); // leggo uno studente
     num++;  // incrementa il numero degli studenti
     }
  num--; // num deve corrispondere al numero totale
  fclose(gt); // chiudo il file
  }
else return (1);  // il file non esiste
```

```
return(0); // il file e' stato caricato
}
```

E' consigliabile il caricamento di una rubrica solo se il programma non ha altri dati al suo interno altrimenti questi potrebbero andare perduti, si consiglia pertanto il salvataggio dei dati presenti oppure lo svuotamento della rubrica.

Queste operazioni hanno complessità pari ad un O(N).

17.8 Programma di Collaudo, salvataggio e caricamento da file

Di seguito e' riportato il programma principale che permette di collaudare ed utilizzare la `Rubrica` presentata in precedenza con le classi `Studente` e `Indirizzo` e con tutte le funzionalità e le versioni degli algoritmi di ricerca che sono state presentate.

Tale classe presenta la seguente definizione:

```
class Rubrica
{
    int num;  // numero degli elementi
    Studente vs[DIMRUB];
public:
     Rubrica(); // costruttore
     void stampa(void); // stampa a video
     Studente * ricercaDT(char * chiave); // ricerca DICOTOMICA
     Studente * ricercaDTR(char * chiave); // ricerca dicotomica
ricorsiva
     Studente * ricerca(char * chiave); // ricerca esaustiva
     int salva(char * namefile);
     int carica(char * namefile);
     int ricercaDTR(int ii, int it, char * chiave);    // passo
elementare della DTR
     int inserimento(Studente &); // inserimento da studente
     void cancella(int pos); // cancellazione
     void svuota(void);
    int cancella(Studente *); // cancellazione
    int modifica(Studente *); // modifica
};



void main(void)
{
char c, buf[DIMSTR];
Rubrica lamia;
Studente stmp, *pstmp; // puntatore a uno studente temporaneo
int ret, flg=0;
```

```
pstmp = new Studente();  // allocazione di uno studente

do {
   printf("\nRubrica, operazioni: \n\n");
   printf("a) inserisci \n");
   printf("b) stampa \n");
   printf("c) ricerca \n");
   printf("f) ricerca Dicotomica \n");
   printf("g) ricerca Dicotomica ricorsiva \n");
   printf("d) cancella \n");
   printf("e) modifica \n");
   printf("l) carica \n");
   printf("m) salva \n");
   printf("n) svuota \n");
   printf("q) esci \n\n");
   printf("Premi un tasto: ");
   c=getch();
   printf("\n");
   switch(c)
      {
      case'a': case 'A':
         stmp.getdati();
         ret=lamia.inserimento(stmp);
         if (ret!=0)
            printf("Rubrica completa ! Operazione impossibile\n");
         break;

      case 'b':  case 'B': // stampa della rubrica
         lamia.stampa();
         break;

      case 'c':  case 'C':
         printf("Studente da cercare, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricerca(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         pstmp->stampa();
         break;

      case 'f':  case 'F': // ricerca dicotomica
         printf("Studente da cercare DT, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricercaDT(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         pstmp->stampa();
         break;
```

```
case 'g':  case 'G': // ricerca dicotomica ricorsiva
   printf("Studente da cercare DT ricorsiva, cognome ?: ");
   scanf("%s", buf);
   pstmp=lamia.ricercaDTR(buf);
   if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
   pstmp->stampa();
   break;

case 'd':  case 'D':
   printf("Studente da cancellare, cognome ?: ");
   scanf("%s", buf);
   pstmp=lamia.ricerca(buf);
   if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
   ret=lamia.cancella(pstmp);
   if (ret!=0) printf("problemi di Cancellazione\n");
   break;

case 'e':  case 'E':
   printf("Studente da modificare, cognome ?: ");
   scanf("%s", buf);
   pstmp=lamia.ricerca(buf);
   if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
   ret=lamia.modifica(pstmp);
   if (ret!=0) printf("problemi di Modifica\n");
   break;

case 'l':  case 'L': // caricamento di un file rubrica
   printf("File Rubrica da Caricare?: ");
   scanf("%s", buf);
   ret=lamia.carica(buf);
     if (ret==1) printf("Il File non esiste \n");
     else if (ret==2) printf ("Svuotare la Rubrica prima \n");
   break;

case 'm':  case 'M': // salvataggio della rubrica in un file
   printf("File Rubrica da Salvare?: ");
   scanf("%s", buf);
   ret=lamia.salva(buf);
     if (ret!=0) printf("Problemi di salvataggio \n");
   break;

case 'n':  case 'N': // svuotare la rubrica
   printf("Svuotare (s/n)?: ");
   c=getch();
   if (c=='s')
         { lamia.svuota();
        printf("Svuotata!\n");
         }
```

```
         break;
      case 'q':  case 'Q':
         flg=1;
         break; // uscita

      default: printf("Operazione non consentita \n"); break;
      }
   if (flg==0)
      { printf("Premi un tasto per un'altra operazione\n");
      getch();
      }
   } while (flg!=1);
}
```

18 Modulo: Associazione, Le Liste di oggetti

Invece di utilizzare un vettore di oggetti e' possibile adottare un'organizzazione dati più flessibile e dinamica: la lista. In questo modo si possono creare delle relazioni 1: N di associazione fra un ogetto ed altri N oggetti.

18.1 Esempio della Rubrica

Se si desidera adottare la struttura a lista per la Rubrica vista in precedenza sono necessarie diverse modifiche. Si noti che nel caso della lista il massimo numero di elementi della struttura dipende dalla disponibilità di memoria e non dal limite imposto sulle dimensioni del vettore come in precedenza.

Il modo più semplice di realizzare una lista e' nel porre direttamente fra gli attributi degli oggetti elementari della lista un attributo puntatore che punti all'elemento successivo in modo da realizzare una struttura a catena.

18.2 La classe Studente come elemento di lista

Il primo passo consiste nel definire una nuova versione della classe `Studente` in modo che le sue istanze possano essere utilizzate in una lista. Per questa ragione, e' stato aggiunto un attributo e due metodi alla classe Studente che e' stata mostrata in precedenza e qui riportiamo per intero per completezza:

```
class Studente
{
     char nome[DIMSTR], cognome[DIMSTR];
     Indirizzo indir;  // indirizzo
     int eta;
     char tel[DIMSTR]; // telefono
     Studente * next;  //-----------Puntatore al successivo
public:
     Studente(void);
     Studente(char *, char *);
     char * get_nome(void);
     char * get_cognome(void);
     char * get_telefono(void );
     int get_eta(void);
     void set_nome_cognome(char * nome, char * cognome);
     void set_eta_telefono(int, char * telefono);
     void set_indirizzo(Indirizzo & );
     void stampa(void);  // stampa dati
     void getdati(void); // richiesta dati da tastiera
    void modifica(void); // modifica
    Studente operator=(Studente &); // assegnazione
    Studente * get_next(void);  // acquisizione del next
```

```
    void set_next(Studente *);  // imposizione del next
};
```

Sono stati anche aggiunti i seguenti metodi selettori per gestire tale puntatore `next`:

```
Studente * Studente::get_next(void) { return (next); }

void Studente::set_next(Studente * a) { next=a; }
```

Anche i costruttori sono stati rivisti per la stessa ragione:

```
Studente::Studente(void) { nome[0]=cognome[0]=tel[0]=eta=0; next=NULL;
}

Studente::Studente(char *a,char *b)
   { strcpy(nome,a); strcpy(cognome,b); tel[0]=eta=0; next=NULL; }
```

18.3 Una Lista di oggetti: la classe RubricaLista

La gestione di una lista di oggetti e' significativamente diversa dalla gestione di un vettore di oggetti. La lista e' molto dinamica e vi e' la necessita' di effettuare un accesso sequenziale ai singoli record.

```
class RubricaLista
{
    int num;  // numero degli elementi
    Studente *start;
public:
     RubricaLista(); // costruttore
     void stampa(void); // stampa a video
     Studente * ricerca(char * chiave); // ricerca esaustiva
     int inserimento(Studente *); // inserimento da studente
     int inserimentoTesta(Studente *); // inserimento da studente
     int cancella(char *); // cancellazione
     int modifica(char *); // modifica
};
```

Risultano pertanto modificati tutti i metodi per la sua gestione sia come parametri che come realizzazione. Anche il costruttore deve tenere conto che non vi e' più un vettore ma un solo puntatore `start` dal quale si diparte la lista di elementi, in questo caso oggetti della classe `Studenti` allocati dinamicamente nella memoria *Heap*.

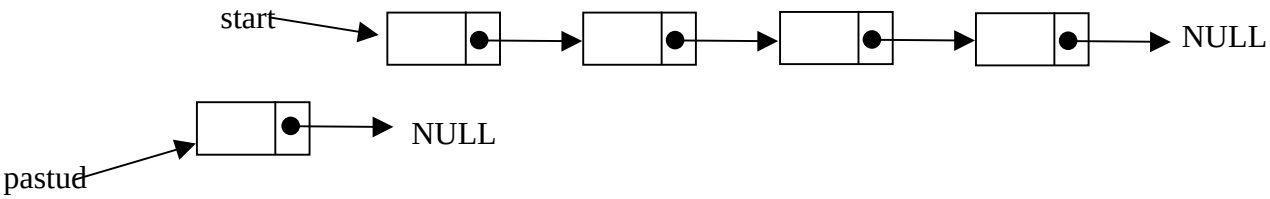
```
RubricaLista::RubricaLista() { num=0; start=NULL; }
```

18.4 Inserimento in una lista ordinata di oggetti: la classe RubricaLista

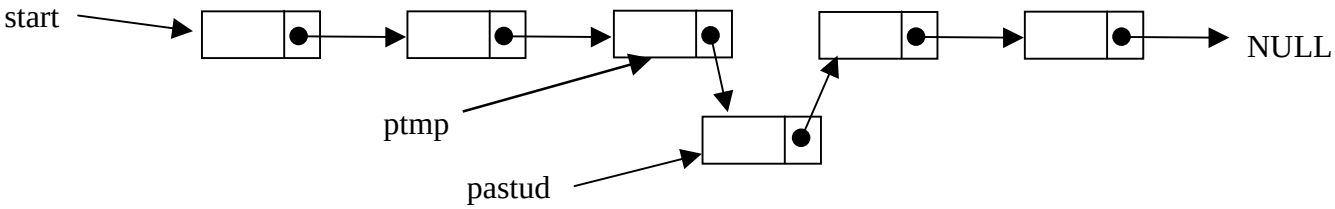
L'operazione di inserimento e' la più delicata quando si lavora con strutture dati ordinate. Senza un inserimento corretto molte delle altre operazioni possono fallire poiché si basano sulla presenza di una struttura con elementi ordinati.

Si deve tenere conto di due casi:

- inserimento in testa e



- l'inserimento nel resto della lista e in coda.



18.5 Inserimento in testa in una lista ordinata di oggetti: la classe RubricaLista

L'inserimento in testa e' un'operazione delicata poiché modifica il valore del puntatore `start` che rappresenta l'unica porta di accesso a tutta l'informazione. La perdita di tale puntatore porta a perdere il riferimento a tutti gli elementi della struttura. Tale situazione e' difficilmente recuperabile. Il metodo che opera l'inserimento in testa viene utilizzato in due occasioni:

1) quando la lista e' vuota, `start==NULL ,`
2) quando l'elemento da inserire ha una chiave di valore precedente a quello del primo elemento della lista, cioè dell'elemento puntato da `start.`

```
int RubricaLista::inserimentoTesta(Studente * pastud)
{
if (pastud==NULL) return(1); // operazione non possibile
if (start==NULL) // inizio di una lista
```

```
  { start = pastud; return(0); }
pastud->set_next(start);
start=pastud; // aggiornamento del puntatore di partenza
return(0);
}
```

L'operazione di inserimento in coda viene automaticamente considerata nella seguente operazione di inserimento generico.

18.6 Inserimento in una lista ordinata di oggetti: la classe RubricaLista

Per le questioni discusse in precedenza la procedura generica di inserimento deve identificare quando vi sono i presupposti per effettuare un inserimento in testa. In tali condizioni l'inserimento viene delegato alla procedura di inserimento in testa precedentemente vista.

Quando si effettua un inserimento si deve comunque controllare che l'elemento non sia presente e valutare la posizione che l'elemento deve avere. A questo fine si effettua una ricerca esaustiva sulla lista ordinata. Se un elemento con lo stesso campo chiave e' già presente nella struttura dati non può essere inserito (non sono previste tecniche per la gestione dei duplicati della chiave). In ogni caso tale elemento può essere inserito utilizzando una chiave anche diversa, per esempio aggiungendo un numero dopo il `cognome`.
Come e' stato mostrato quando sono state esposte le strutture dati, per effettuare un inserimento e' necessario avere il puntatore all'elemento precedente alla posizione nella quale si intende inserire il nuovo elemento. Nella procedura seguente tale informazione viene acquisita con una ricerca e posta nel puntatore temporaneo a `Studente ptmp`.

```
int RubricaLista::inserimento(Studente * pastud)
{ int ret;
Studente *ptmp=NULL;

if (start==NULL)  // lista vuota
  return(inserimentoTesta(pastud) );

ptmp=start;
ret = strcmp(ptmp->get_cognome(), pastud->get_cognome() );
if (ret==0)
  { printf("Studente gia' presente \n");
  ptmp->stampa(); return(1);
  }
else if (ret>0) // studente da inserire prima del primo
  return( inserimentoTesta(pastud) );

// ricerca se lo studente e' gia' inserito e della posizione
while (ptmp->get_next()!=NULL)
  { ret=strcmp((ptmp->get_next())->get_cognome(),pastud-
>get_cognome());
  if (ret==0)
```

```
    { printf("Studente gia' presente \n");
    (ptmp->get_next())->stampa();
    return(1);
    }
  else if (ret>0) break; // posizione di inserimento trovata
  ptmp=ptmp->get_next(); // passaggio al successivo
  }
// ptmp indica l'oggetto prima della posizione di inserimento
// inserimento
pastud->set_next(ptmp->get_next());
ptmp->set_next(pastud);

// aggiornamento del valore di num
num++;
return(0);
}
```

Nei casi in cui l'inserimento non e' possibile viene restituito un valore non zero. Tale valore viene utilizzato dal programma principale per segnalare l'errore

18.7 Stampa di una lista di oggetti: la classe RubricaLista

L'operazione di stampa degli elementi di una lista di oggetti e' piuttosto semplice si tratta di navigare sulla lista per mezzo del puntatore all'elemento successivo per arrivare dal primo all'ultimo elemento. Questa operazione globale sulla lista ha una complessità asintotica pari ad un O(N). Questa operazione non dipende dal criterio di ordinamento utilizzato, la lista potrebbe anche non essere ordinata.
L'operazione di stampa non dipende dai dati pertanto non vi sono condizioni migliori peggiori e medie, il numero di operazioni e' sempre un O(N).

La stampa si interrompe quando si arriva all'ultimo elemento della lista. Questo può essere facilmente riconosciuto poiché presenta un valore `NULL` sul suo `next`.
Poiché la lista potrebbe essere molto lunga, e' stata inserita la richiesta di pressione di un tasto per procedere a visualizzare l'elemento successivo con la funzione `getch()`.

```
void RubricaLista::stampa(void) // stampa a video
{ Studente *ptmp=start;
int i=0;
while (ptmp!=NULL)
  { printf("--------Studente %d ----------\n", i);
  ptmp->stampa();
  ptmp=ptmp->get_next();
  printf("Premi un tasto per continuare \n");
  getch();
  }
}
```

18.8 Ricerca su una lista ordinata di oggetti: la classe RubricaLista

L'operazione di ricerca di un elemento di una lista ordinata di oggetti in base ad un campo chiave e' piuttosto semplice si tratta di navigare sulla lista per mezzo del puntatore all'elemento successivo per arrivare dal primo all'ultimo elemento. Questa operazione globale sulla lista ha un complessità asintotica pari ad O(N). Nel caso migliore l'operazione termina dopo il primo confronto. Il caso migliore si ha quando l'elemento cercato e' il primo. Il caso peggiore quando l'elemento cercato e' l'ultimo, il caso medio quando si trova nel mezzo della lista e quindi si devono effettuare N/2 passi per trovarlo.

La ricerca si interrompe quando si trova l'elemento cercato oppure si arriva all'ultimo elemento della lista. Questo può essere facilmente riconosciuto poiché presenta un valore NULL sul suo `next`. Per identificare l'elemento cercato viene effettuato un confronto fra stringhe.

```
Studente * RubricaLista::ricerca(char * chiave)   // ricerca esaustiva
{ Studente *ptmp=start;
while (ptmp!=NULL)
  { if (strcmp(ptmp->get_cognome(), chiave)==0)
      return ( ptmp );
  ptmp=ptmp->get_next();
  }
return(NULL);
}
```

Tale valore `NULL` viene utilizzato dal programma principale per segnalare l'errore.

18.9 Ricerca dicotomica su lista ordinata

La ricerca dicotomica e' un algoritmo che si basa sull'accesso diretto agli oggetti e sulla struttura lineare della memoria. E' tramite tali ipotesi che e' possibile stimare l'indice dell'elemento centrale che divide il vettore/sottovettore in due parti.

Nelle liste questi presupposti non sono soddisfatti e pertanto la realizzazione della ricerca dicotomia risulta difficile e non efficiente. Tale realizzazione potrebbe basarsi sulla posizione nella lista, dove per posizione si intende un numero che indichi la distanza dell'elemento dalla radice.

In questo modo la funzione per trovare il punto di mezzo potrebbe fornire la posizione mediana. Ad ogni calcolo della posizione mediana e' pero' necessario navigare sulla lista per raggiungere l'elemento questo comporta una complessità pari ad un O(N). Si viene pertanto ad avere una complessità che e' almeno lineare invalidando i benefici della ricerca dicotomica che presenta una complessità asintotica pari a O(Log N).

18.10 Cancellazione su una lista ordinata di oggetti: la classe RubricaLista

Per la cancellazione si deve effettuare un procedimento simile a quello adottato nell'inserimento. In questo caso vi sono sostanzialmente due casi:

1) cancellazione del primo elemento, quello puntato da `start`;
2) cancellazione degli altri elementi, incluso l'ultimo della lista.

Il metodo che segue comprende tutti e due questi casi. Prima si verifica che l'elemento da cancellare non sia il primo. In caso affermativo si effettua la sua cancellazione modificando il valore di `start` e si conclude il metodo con un `return().` Prima di uscire dal metodo si aggiorna la variabile `num` che tiene conto di quanti elementi sono presenti nella lista. Tale variabile non ha altra funzionalità che tenere il numero.

```
int RubricaLista::cancella(char * chiave) // cancellazione
{ int ret;
Studente *ptmp=NULL, *prevptmp;

ptmp=start;
ret = strcmp(ptmp->get_cognome(), chiave);
if (ret==0) // cancellazione in testa
  { start=start->get_next(); // aggiornamento dello start
  delete(ptmp);  // deallocazione
  num--; // aggiornamento del numero
  return(0);
  }
prevptmp=ptmp; // aggiornamento del puntatore precedente
ptmp=ptmp->get_next(); // passaggio al successivo

while (ptmp!=NULL)
  { if ( strcmp( ptmp->get_cognome(), chiave)==0 ) // trovato
    { prevptmp->set_next(ptmp->get_next()); // ristabilisce i legami
    delete(ptmp);  // deallocazione
    num--; // aggiornamento del numero
    return(0);
    }
  prevptmp=ptmp;  // aggiornamento del puntatore precedente
  ptmp=ptmp->get_next(); // passaggio al successivo
  }
return(1);
}
```

Nella seconda parte del metodo si ha una ricerca dell'elemento da cancellare e quindi la sua cancellazione vera e propria. Si noti che la cancellazione non implica una riorganizzazione pesante come nei vettori. In questo caso e' necessario ripristinare la catena senza l'elemento che si intende cancellare dopo di che si può procedere alla sua deallocazione dalla memoria *heap* con l'uso della funzione `delete()`.

Se non viene cancellato nessun elemento significa che nella lista non vi sono elementi con tale chiave e pertanto il metodo ritorna un valore non zero. Tale valore viene utilizzato dal programma principale per segnalare l'errore.

18.11 Modifica su una lista ordinata di oggetti: la classe RubricaLista

La modifica del contenuto informativo di un oggetto non e' un problema se non viene modificato il campo chiave. Quando si ha una modifica del campo chiave, in generale l'oggetto non risulta più nella posizione corretta rispetto agli altri elementi in base al criterio di ordinamento utilizzato per la struttura dati. Se l'operazione di modifica del cambio chiave e' necessaria deve essere possibile vedere se vi e' stato un cambio di chiave. In tale caso, si può salvare l'oggetto, cancellarlo dalla struttura e reinserirlo nella giusta posizione.

Di seguito viene riportato il metodo modifica per la classe `RubricaLista` che permette anche la modifica del campo chiave.

In questo caso nel metodo modifica e' inclusa la parte di ricerca dell'elemento da modificare.
E' possibile realizzare una procedura di modifica più efficiente di questa, un metodo nel quale la cancellazione non dealloca l'oggetto ma semplicemente lo estrae dalla lista per poi re-inserirlo. In questo caso non e' stata adottata tale soluzione poiché si e' tratto vantaggio dalla presenza del metodo `cancella().`

Comunque le operazioni di cambio chiave sono abbastanza rare e pertanto una procedura non ottimizzata non pregiudica le prestazioni della soluzione.

```
int RubricaLista::modifica(char * chiave) // cancellazione
{
Studente *ptmp=start, *std;
char nometmp[DIMSTR];
int ret;

while (ptmp!=NULL) // ciclo di ricerca
  {
  if (strcmp(ptmp->get_cognome(), chiave)==0)  // oggetto trovato
    {
    strcpy(nometmp, ptmp->get_cognome() ); // salvo la chiave
    ptmp->modifica();       // modifica dello studente

    if ( strcmp(nometmp, ptmp->get_cognome())!=0 ) // cambio chiave ?
      {
      std = new Studente();  // alloco un nuovo studente
      *std = *ptmp;          // ci copio il vecchio valore
      std->set_next(NULL);   // impongo nullo il suo next
      cancella(ptmp->get_cognome()); // cancello lo studente fuori
posto
      ret=inserimento(std);  // inserisco il nuovo studente
      if (ret!=0) return(1);
      }
    return(0); // operazione di modifica con successo
    }
```

```
  ptmp=ptmp->get_next(); // passaggio al successivo
  }
return(1); // non trovato
}
```

Nei casi in cui la modifica non e' possibile viene restituito un valore non zero. Tale valore viene utilizzato dal programma principale per segnalare l'errore

18.12 La classe RubricaLista: programma di collaudo

Di seguito e' riportato il programma principale che permette di collaudare ed utilizzare la classe `RubricaLista` presentata in precedenza con le classi `Studente` e `Indirizzo`. Si noti la realizzazione di un piccolo menu di scelte effettuato per mezzo di alcune chiamate alla funzione `printf()` e della funzione `getch()` per acquisire la scelta. I vari casi, contengono anche le istruzioni per gestire i valori di ritorno di vari metodi della classe `RubricaLista`. Si noti come le operazioni di cancellazione e modifica non necessitano di una ricerca per identificare l'oggetto.

```
void main(void)
{
char c, buf[DIMSTR];
RubricaLista lamia;
Studente stmp, *pstmp; // puntatore a uno studente temporaneo
int ret, flg=0;

pstmp = new Studente();

do { // menu
   printf("\nRubricaLista, operazioni: \n\n");
   printf("a) inserisci \n");
   printf("b) stampa \n");
   printf("c) ricerca \n");
   printf("d) cancella \n");
   printf("e) modifica \n");
   printf("q) esci \n\n");
   printf("Premi un tasto: ");
   c=getch();
   printf("\n");

   switch(c)
      { case'a': case 'A':
         pstmp= new Studente();
         if (pstmp==NULL)
            printf("Operazione impossibile\n");
         else
           { pstmp->getdati();
           ret=lamia.inserimento(pstmp);
           if (ret!=0) printf("Operazione impossibile\n");
```

```
            }
         break;

      case 'b':  case 'B':
         lamia.stampa(); break;


      case 'c':  case 'C':
         printf("Studente da cercare, cognome ?: ");
         scanf("%s", buf);
         pstmp=lamia.ricerca(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         pstmp->stampa();
         break;

      case 'd':  case 'D':
         printf("Studente da cancellare, cognome ?: ");
         scanf("%s", buf);
         ret=lamia.cancella(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         break;

      case 'e':  case 'E':
         printf("Studente da modificare, cognome ?: ");
         scanf("%s", buf);
         ret=lamia.modifica(buf);
         if (pstmp==NULL) { printf("Studente non trovato\n"); break; }
         break;

      case 'q':  case 'Q':
         flg=1;
         break; // uscita

      default: printf("Operazione non consentita \n"); break;
      }
   if (flg==0)
      { printf("Premi un tasto per un'altra operazione\n");
      getch();
      }
   } while (flg!=1);
}
```

18.13 Complessità nella gestione di liste ordinate

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di un certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni.

- Per inserimento si intende l'operazione di inserzione senza considerare la ricerca ma considerando eventuali operazioni di riorganizzazione.
- Per cancellazione si intende la sola operazione di eliminazione dell'oggetto senza considerare la ricerca ma considerando le eventuali operazioni di riorganizzazione.
- Per modifica si intende la sola operazione di modifica dell'oggetto senza considerare la ricerca e senza considerare il caso in cui si ha un cambio chiave e quindi si deve cancellare e reinserire.

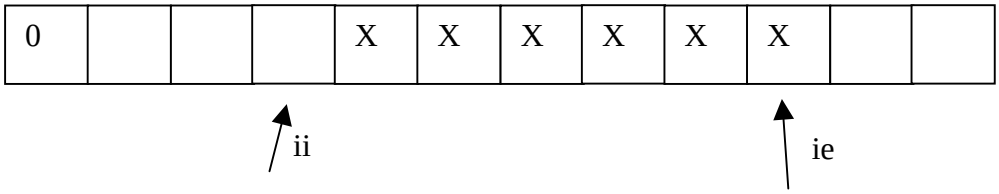
Nella seguente tabella sono riportate le complessità asintotica, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su una lista ordinata di N oggetti (per esempio una lista di studenti nella classe `RubricaLista)`. Fra queste operazioni l'unica globale e' quella di stampa, le altre sono operazioni sostanzialmente sul singolo oggetto.

Lista ordinata di N elementi				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(1)	1	1	1
Cancellazione	O(1)	1	1	1
Ricerca	O(N)	N/2	1	N
Modifica	O(1)	1	1	1
Stampa	O(N)	N	N	N

19 Modulo: Il Buffer o coda

19.1 Buffer realizzato con un vettore: classe Buffer

La realizzazione di un buffer tramite un vettore e' molto semplice. Per tale scopo si utilizzano due indici, `ii` come indice di inserimento (indica la posizione che deve assumere il prossimo elemento che viene inserito), ed `ie` come indice di estrazione (indica il primo elemento da estrarre). Gli elementi presenti sono marcati con una X.



Nell'esempio codificato in C++ che segue e' stato realizzato un buffer di variabili `float` di dimensione `DIMBUF` pari a 10, in accordo al `define`.

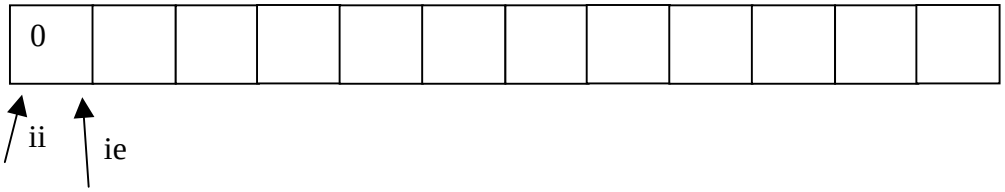
```
#define DIMBUF 10

class Buffer
{
     int ii; // indice di inserimento (posizione in cui inserire)
     int ie; // indice di estrazione (posizione dell'elemento da
estrarre)
             // se ii==ie e' vuoto
     float corpo[DIMBUF];
public:
     Buffer();
     int inserimento(float a);  // inserisci un elemento
     int estrazione(float * a); // estrai un elemento
     void svuota(void);  // svuota il buffer
     void stampa(void);  // stampa il buffer
     int getnumero(void); // numero di elementi
};
```

All'inizializzazione il buffer e' vuoto e quindi i valori degli indici sono impostati a DIMBUF-1;

```
Buffer::Buffer()
{ int i;   ii=ie=DIMBUF-1;   for (i=0;i<DIMBUF;i++) corpo[i]=0.0;  }
```

Ogni *inserimento* provoca l'aggiunta di un elemento nel vettore e lo spostamento verso l'elemento di indice zero dell'indice di inserimento `ii`.
Ogni *estrazione* comporta lo spostamento dell'indice di estrazione, `es`, di un posto verso l'elemento di indice zero. L'estrazione può essere effettuata solo se l'indice di estrazione non e' uguale a quello di inserimento. Tale situazione stabilisce che il buffer e' vuoto.

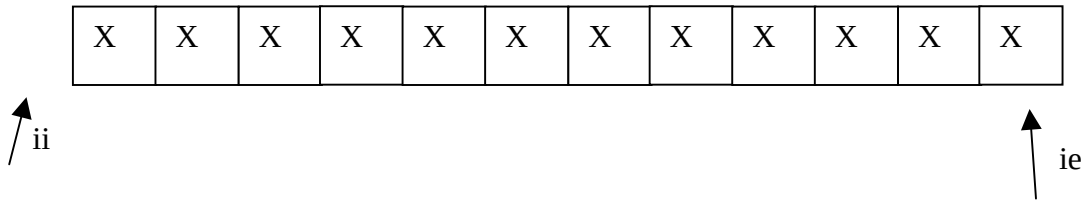
In sostanza, ogni operazione porta gli indici verso lo zero del vettore e quindi dopo un certo numero di operazioni di inserimento il buffer si riempie e anche se queste sono compensate da un equivalente numero di estrazioni, il buffer non e' in grado di accettare altri inserimenti come riportato in figura.



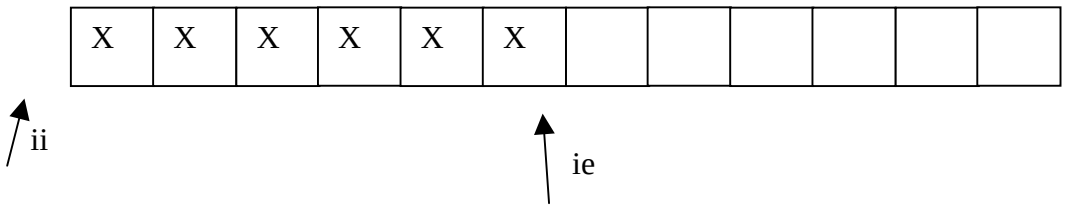
Per uscire da questa situazione e' necessario effettuare una riorganizzazione del buffer per riportare il buffer nelle condizioni iniziali.

Questa riorganizzazione può essere effettuata in due occasioni:
1. Ad ogni inserimento in modo da portare l'indice di estrazione allineato all'ultimo elemento come nelle condizioni iniziali e quello di inserimento il piu' avanti possibile.
2. Ad ogni estrazione in modo da portare sempre l'indice di estrazione allineato all'ultimo elemento come nelle condizioni iniziali.

Nel primo caso si riscontra un malfunzionamento. Per esempio se il buffer viene riempito arrivando alla situazione in figura.



E quindi si operano delle operazioni di estrazione, portando il buffer alla situazione che segue, non e' piu' possibile aggiungere elementi anche se in effetti vi e' lo spazio. Il buffer risulta bloccato.



Nel secondo caso si ha un funzionamento migliore, ogni qual volta si viene a creare dello spazio per mezzo di un' estrazione questo viene reso disponibile per l'inserimento. L'estrazione viene ad avere una complessità asintotica pari ad un O(N).

Esercizio: Si verifichi la condizione di stallo e si realizzi i metodi di inserimento e estrazione in base al secondo caso, in modo da evitare la situazione di stallo con il buffer non pieno.

Una soluzione per ovviare al problema del primo caso consiste nel verificare ad ogni inserimento la possibilità di effettuare una riorganizzazione. Questo limita il numero delle riorganizzazioni a quelle strettamente necessarie. Se il buffer viene utilizzato facendo inserimenti e estrazioni in modo alternato si arriva ad uno stato di regime nel quale ad ogni inserimento si ha una riorganizzazione ogni M inserimenti, dove M e' pari a N-E, con E il numero di elementi nel buffer e N la dimensione del vettore.

Di seguito viene riportata la realizzazione dei metodi di inserimento e estrazione per la classe Buffer. Tali metodi fanno riferimento a questo ultimo caso.

Il principale limite dei Buffer realizzati con i vettori e' la staticità della struttura. Un vettore ha una dimensione fissa che non può essere modificata per fare fronte alle necessita'.

19.2 Inserimento in un buffer realizzato con un vettore: classe Buffer

Procedura di inserimento secondo il primo modello di gestione del buffer. Si noti la parte in cui viene effettuata la riorganizzazione del buffer. Se `ii` e' posizionato all'inizio del buffer impedendo l'inserimento di altri elementi viene verificata la possibilità di predisporre altro spazio riorganizzando il buffer, spostando i dati verso la fine del vettore.

```
int Buffer::inserimento(float a)  // inserisci un elemento
{ int i;
if (ii<0)
  { // necessita riorganizzazione se non e' pieno
  if (ie==DIMBUF-1) return(1); // pieno
  for (i=ie;i>=0;i--)// riorganizzazione
     corpo[DIMBUF-1+i-ie]=corpo[i];
  ii=DIMBUF-1-ie-1; // reimpostazione degli indici
  ie=DIMBUF-1;
  for (i=0;i<=ii;i++) corpo[i]=0.0; // cancellazione delle celle
vecchie
  }
corpo[ii]=a;
ii--;
return(0);
}
```

Nell'algoritmo sono cancellate anche le celle che nel corso della riorganizzazione non hanno piu' validità poiché il loro valore e' stato copiato piu' avanti. Tale operazione può anche non essere effettuata.

19.3 Estrazione da un buffer realizzato con un vettore: classe Buffer

In accordo a quanto detto in precedenza per il metodo di inserzione, il metodo di estrazione non presenta la riorganizzazione che e' appunto delegata al metodo di inserzione.

La complessità dell'operazione di estrazione e' pertanto unitaria.

```
int Buffer::estrazione(float * a) // estrai un elemento
{
if (ie>=0 && ii!=ie)
  { *a=corpo[ie];  // copia dell'elemento sul puntatore esterno
  corpo[ie]=0.0;   // cancellazione dell'elemento estratto
  ie--;            // decremento del numero di elementi
  return(0);       // uscita dalla funzione
  }
return(1);
}
```

19.4 Svuotamento e stampa di un buffer realizzato con un vettore: classe Buffer

In questa sessione sono riportati alcuni metodi di servizio.
Il seguente metodo permette di svuotare completamente un buffer riportandolo alle condizioni iniziali.

```
void Buffer::svuota(void)  // svuota buffer
  { int i;   ii=ie=DIMBUF-1;
  for (i=0;i<DIMBUF;i++) corpo[i]=0.0;
  }
```

La stampa del buffer e' un'operazione tipicamente non funzionale alla gestione del buffer ma che può essere utile per capire lo stato del buffer durante il suo sviluppo e allo studente che voglia osservare lo stato dopo ogni cambiamento.

```
void Buffer::stampa(void)  // svuota buffer
{
int i;
for (i=0;i<DIMBUF;i++) printf("%f, ",corpo[i]);
printf("\n");
}
```

Il seguente metodo permette di conoscere il numero di elementi contenuti nel buffer.

```
int Buffer::getnumero(void) { return(ie-ii); }
```

19.5 Programma di test per il buffer realizzato con un vettore: classe Buffer

In questa sessione e' riportato un programma di collaudo per la classe Buffer descritta in precedenza. Tale programma permette l'inserimento di elementi, la loro estrazione e la visione dello stato del buffer.

```
---premi qui per provare il programma---

void main(void)
{ char c;
int ret, flg=0;
float val;
Buffer buf; // dichiarazione di un buffer

do {
   printf("\nBuffer, operazioni: \n\n");
   printf("a) inserisci \n");
   printf("b) estrai \n");
   printf("c) stampa \n");
   printf("d) svuota \n");
   printf("e) numero \n");
   printf("q) esci \n\n");
   printf("Premi un tasto: ");
   c=getch();
   printf("\n");

   switch(c)
      { case'a': case 'A':
         printf("valore da inserire %f:"); scanf("%f",&val);
         ret=buf.inserimento(val);
         if (ret!=0) printf("Buffer pieno!\n");
         break;
      case 'b':  case 'B':
         ret=buf.estrazione(&val);
         if (ret!=0) printf("Buffer vuoto!\n");
         else printf("Valore estratto %f\n", val);
         break;
      case 'c':  case 'C':
         printf("Stampa \n"); buf.stampa(); break;
      case 'd':  case 'D':
         printf("Svuota \n"); buf.svuota(); break;
      case 'e':  case 'E':
         ret=buf.getnumero();
         printf("Il buffer contiene %d elementi \n", ret);
         break;
      case 'q':  case 'Q':
         flg=1; break; // uscita
```

```
      default: printf("Operazione non consentita \n"); break;
      }
   if (flg==0)
      { printf("Premi un tasto per un'altra operazione\n");
      getch();
      }
   } while (flg!=1);
}
```

19.6 Complessità nella gestione del buffer realizzato con un vettore

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di un certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni. Nel caso di un buffer realizzato con un vettore:

- Per inserimento si intende l'operazione di inserzione considerando eventuali operazioni di riorganizzazione.
- Per estrazione si intende la sola operazione di eliminazione dell'oggetto considerando le eventuali operazioni di riorganizzazione.

Nella seguente tabella sono riportate le complessità asintotiche, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su un buffer di N oggetti realizzato con un vettore.

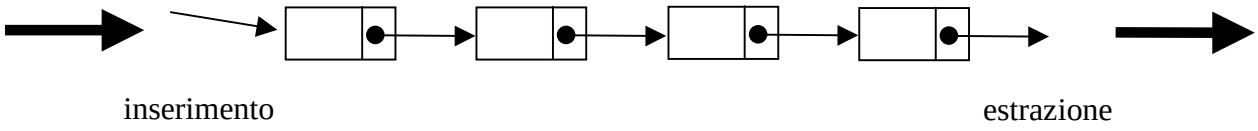
Buffer con vettore di N elementi				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(N)*	N*/2	1	N
Estrazione	O(1)	1	1	1
Stampa	O(N)	N	N	N
Svuotamento	O(N)	N	N	N

Si ricorda che in fase di inserimento non si ha sempre da riorganizzare il buffer, questo dipende dalle dimensioni del buffer e dalla quantità di elementi che contiene in media.

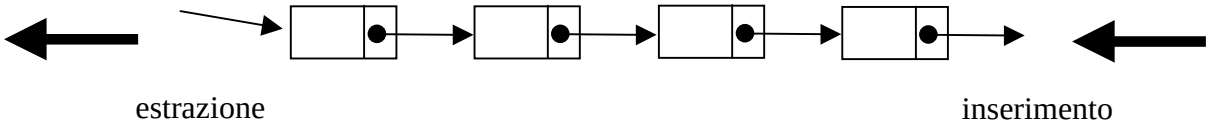
19.7 Classe Buffer realizzata da un lista

In questo caso si possono avere due diverse implementazioni a seconda che si operi una soluzione per la quale si adotta:

1) inserimento in testa ed estrazione in coda,

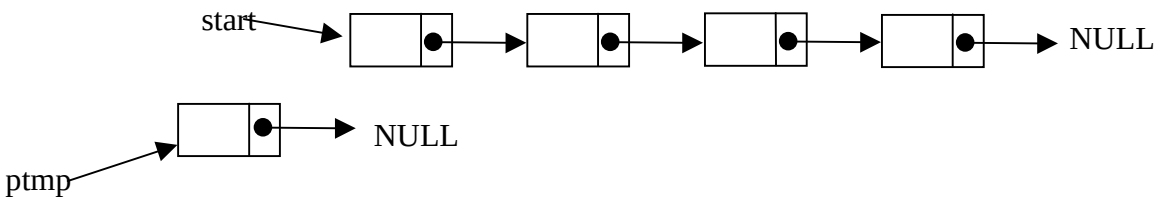


2) inserimento in coda ed estrazione in testa.

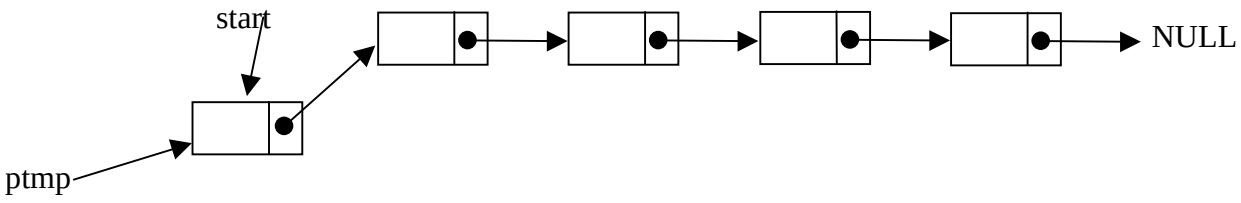


19.8 Lista: Inserimento in testa

L'operazione di inserimento in testa e' molto semplice. La situazione di partenza e' la seguente in cui si ha una lista referenziata dal puntatore `start` e un elemento nuovo da inserire referenziato da `ptmp`:

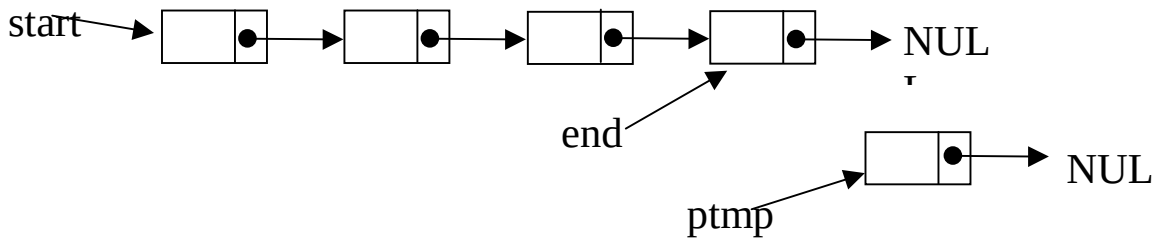


Il primo passo e' di collegare il nuovo elemento alla lista con: `ptmp->get_next(start);` e quindi reimpostare il valore di `start` a `ptmp`.

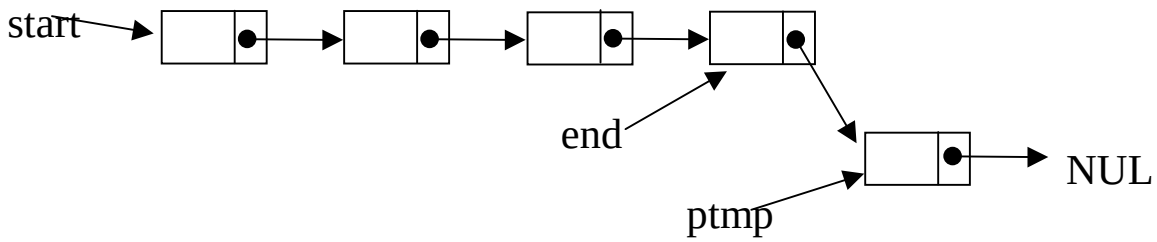


19.9 Lista: Inserimento in coda

L'operazione di inserimento in coda e' abbastanza semplice. Di una lista tipicamente se ne conosce il puntatore alla testa, cioè `start` mentre non si conosce il puntatore all'ultimo elemento, posizione dalla quale si può operare per effettuare un inserimento in coda. Pertanto al fine di rendere possibile tale operazione e' necessario conoscere oltre al puntatore `start` anche un puntatore all'ultimo elemento(chiamato in seguito `end`). Per effettuare un inserimento in coda si può partire dalla seguente condizione:

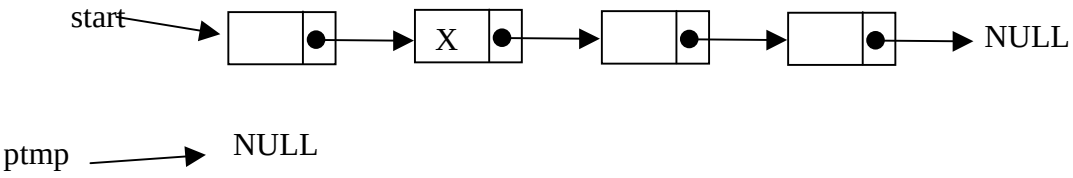


Per effettuare l'inserimento e' sufficiente collegare la lista al nuovo elemento con:
`end->set_next(ptmp);` e verificando che il puntatore al `next` dell'elemento inserito sia `NULL` poiché questo diventa il nuovo terminatore della lista, ottenendo il seguente risultato:

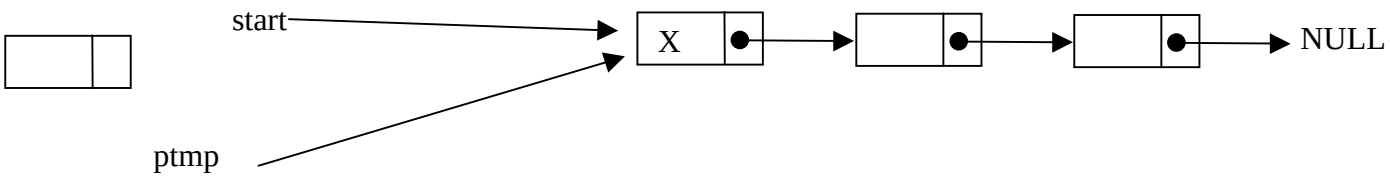


19.10 Lista: Estrazione in testa

L'operazione di estrazione dell'elemento che si trova in testa alla coda e' abbastanza semplice se si conosce il puntatore alla testa della coda. Da tale puntatore si può arrivare a conoscere il puntatore all'elemento successivo al primo (marcato con X nella figura) che deve diventare dopo l'estrazione il primo elemento della lista. A questo scopo si può utilizzare un puntatore temporaneo `ptmp`. La condizione di partenza e' la seguente:

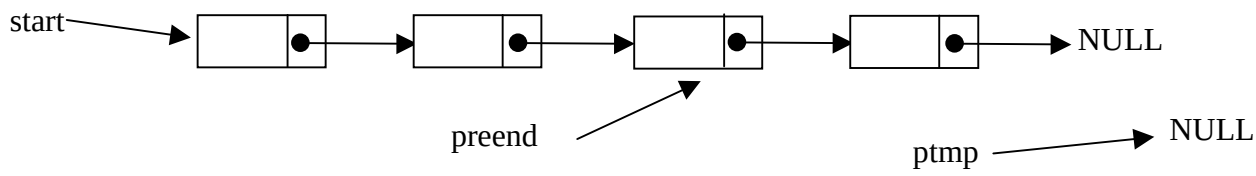


Da tale condizione con l'istruzione: `ptmp->set_next(start->get_next())` si acquisisce il puntatore all'elemento marcato con X e lo si assegna a `ptmp`. A questo punto il contenuto informativo del primo elemento (puntato ancora da `start`) può essere consumato, copiato, connesso con un altro puntatore, o cancellato (`delete(start)`). Questo dà la possibilità di riutilizzare `start` per memorizzare l'inizio della lista con: `start=ptmp;`



19.11 Lista: Estrazione in coda

L'operazione di estrazione di un elemento dalla fine della lista non e' semplice dal punto di vista computazionale. Per eliminare l'ultimo elemento della coda e' necessario avere a disposizione il puntatore all'elemento precedente. Questo e' possibile utilizzando un puntatore al penultimo, per esempio `preend`. Tale condizione non può pero' essere mantenuta con un costo computazionale contenuto. Ogni qual volta che si effettua una estrazione dalla coda tale puntatore al penultimo deve essere ricalcolato poiché non e' possibile navigare indietro sulla lista. Tale operazione può essere svolta solamente ripercorrendo la lista dall'inizio, con un costo computazionale pari ad un O(N).

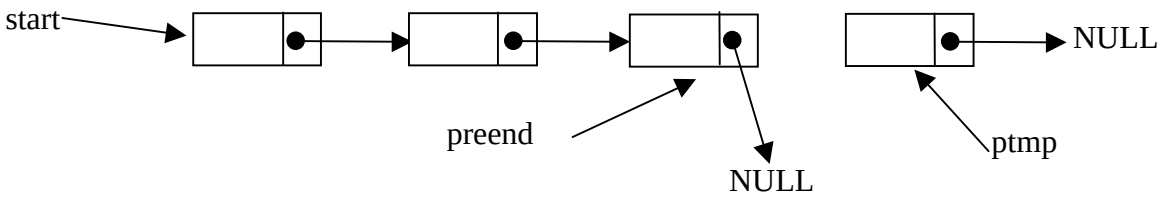


Data la presenza di `preend` si opera l'estrazione utilizzando un puntatore temporaneo `ptmp` per referenziare l'oggetto estratto:

`ptmp->set_next(preend->get_next());` // assegnazione di `ptmp` all'ultimo elemento
`preend->set_next(NULL);` // terminazione della lista

Si poteva anche eliminare l'ultimo con `delete(preend->get_next());`

In questo caso si opera direttamente la deallocazione dell'ultimo elemento.



19.12 Complessità nella gestione del buffer realizzato con una lista

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di un certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni. Nel caso di un buffer realizzato con una lista:

- Per inserimento si intende l'operazione di inserzione considerando eventuali operazioni di riorganizzazione.
- Per estrazione si intende la sola operazione di eliminazione dell'oggetto considerando le eventuali operazioni di riorganizzazione.

Nella seguente tabella sono riportate le complessità asintotica, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su un buffer di N oggetti realizzato con una lista concatenata monodirezionale.

In questo caso si possono avere due diverse implementazioni a seconda che si operi una soluzione per la quale si adotta:

1) inserimento in testa ed estrazione in coda,

2) Inserimento in coda ed estrazione in testa.

Buffer con lista modirezionale di N elementi				
Inserimento in testa, estrazione in coda				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(1)	1	1	1
Estrazione	O(N)	N	N	N
Stampa	O(N)	N	N	N
Svuotamento	O(N)	N	N	N

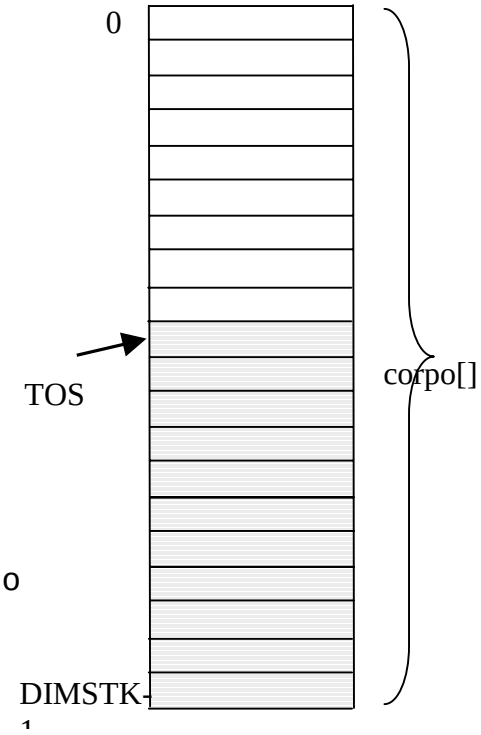
Buffer con lista modirezionale di N elementi				
Inserimento in coda, estrazione in testa				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento	O(1)	1	1	1
Estrazione	O(1)	1	1	1
Stampa	O(N)	N	N	N
Svuotamento	O(N)	N	N	N

E' evidente che la soluzione più conveniente dal punto di vista computazionale e' quella che vede l'inserimento in coda e l'estrazione in testa.

20 Modulo: Lo Stack o coda

20.1 Stack con vettore: classe Stack

La realizzazione di uno stack con un vettore e' abbastanza semplice. La classe Stack deve avere fra i propri attributi un vettore di elementi ed un indice, il TOS (Top of the Stack). Sono inoltre inseriti nella classe i metodi `push()`, `pop()`, `isempty()` e `stampa()`.




```
#define DIMSTK 10
class Stack
{
     int tos; // top of stack
     float corpo[DIMSTK];
public:
     Stack();
     void svuota(void); // svuota lo stack
     int push(float a); // inserisci un elemento
     int pop(float * a); // estrai un elemento
     int isempty(void); // verifica se e' vuoto
     void stampa(void); // stampa
};
```

Lo stack viene realizzato per mezzo di un vettore di dimensione `DIMSTK`. Pertanto il primo elemento viene inserito nella cella `DIMSTK-1`, come riportato in figura. Il `tos` indica il primo dato che puo' essere estratto dallo stack. Il contenuto dello stack viene fatto crescere per mezzo di operazioni di `push()` a partire dalla fine del vettore. Quando il vettore si svuota oppure quando e' pieno si deve segnalare l'errore. Per semplicita' e' stato realizzato uno stack di `float`.

20.2 Stack, metodi costruttore, svuota() e push(), classe Stack

Alla sua creazione e allo svuotamento lo stack viene inizializzato in modo opportuno ponendo il `tos` all'ultimo elemento del vettore e cancellando il contenuto degli elementi del vettore.

```
Stack::Stack()
  { int i;
  tos=DIMSTK-1;
  for (i=0;i<DIMSTK;i++) corpo[i]=0.0;
  }

void Stack::svuota(void) { Stack(); }
```

L'operazione di `push()` può essere eseguita solo se lo stack non e' vuoto. In tal caso viene inserito il nuovo elemento ed aggiornato il valore del `tos` decrementandolo.

```
int Stack::push(float a)  // inserisci un elemento
{
if (tos>=0) // controllo per stack pieno
  {
  corpo[tos]=a;  // inserimento dell'elemento
  tos--;   // descremento del tos
  return(0);
  }
return(1);
}
```

20.3 Stack, metodi pop(), stampa() ed isempty(), classe Stack

L'operazione di `pop()` può essere eseguita solo se lo stack non e' vuoto. In tal caso viene inserito il nuovo elemento ed aggiornato il valore del `tos` decrementandolo.

```
int Stack::pop(float * a) // estrai un elemento
{ if (tos<DIMSTK-1)
  { tos++; *a = corpo[tos];
  corpo[tos]=0.0; // cancellazione del contenuto [opzionale]
  return(0);
  }
return(1);
}
```

La stampa non e' un metodo necessario al funzionamento dello stack ma puo' essere utile per visualizzare lo stato dello stack al fine di capirne il suo funzionamento.

```
void Stack::stampa(void)
{int i;
printf("-------Stack--------\n");
for (i=0;i<DIMSTK;i++) printf("%d) %f\n",i, corpo[i]);
printf("tos %d \n", tos);
}
```

Questo metodo permette di controllare lo stato dello stack.

```
int Stack::isempty(void)
  { if (tos==DIMSTK-1) return(1); else return(0); }
```

20.4 Complessità nella gestione dello *Stack* realizzato con un vettore

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di un certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni. Nel caso di uno *stack* realizzato con un vettore:

- Per inserimento si intende l'operazione di inserzione/*push* considerando eventuali operazioni di riorganizzazione.
- Per estrazione si intende la sola operazione di eliminazione/*pop* dell'oggetto considerando le eventuali operazioni di riorganizzazione.

Nella seguente tabella sono riportate le complessità asintotica, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su uno *Stack* di N oggetti realizzato con un vettore.

Stack con vettore di N elementi				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Inserimento/push	O(1)	1	1	1
Estrazione/pop	O(1)	1	1	1
Stampa	O(N)	N	N	N
Svuota	O(N)*	N*	N*	N*
Isempty	1	1	1	1

Lo svuotamento può essere effettuato in modo semplice imponendo solo il valore del `tos` senza cancellare i valori contenuti nel vettore, in tale caso il costo e' unitario.

20.5 Programma di collaudo per la Classe Stack realizzata come vettore

In questa sessione e' riportato un programma di collaudo per la classe Stack descritta in precedenza. Tale programma permette l'inserimento di elementi, la loro estrazione e la visione dello stato dello Stack, nonché il suo svuotamento.

```
---premi qui per provare il programma---

void main(void)
{
char c;
int ret, flg=0;
float val;
Stack stk; // oggetto di tipo stack

do {
   printf("\nStack, operazioni: \n\n");
   printf("a) push \n");
   printf("b) pop \n");
   printf("c) stampa \n");
   printf("d) isempty \n");
   printf("e) svuota \n");
   printf("q) esci \n\n");
```

```
   printf("Premi un tasto: ");

   c=getch();      // acquisizione della scelta
   printf("\n");

   switch(c)
      { case'a': case 'A':   // inserimento/push
         printf("valore da inserire %f:"); scanf("%f",&val);
         ret=stk.push(val);
         if (ret!=0) printf("Stack pieno!\n");
         break;

      case 'b':  case 'B':  // estrazione/ pop
         ret=stk.pop(&val);
         if (ret!=0) printf("Stack vuoto!\n");
         else printf("Valore estratto %f\n", val);

         break;
      case 'c':  case 'C':  // stampa
         printf("Stampa \n");
         stk.stampa();
         break;

      case 'd':  case 'D': // controllo per vuoto
         ret=stk.isempty();
         if (ret) printf("Lo stack e' vuoto \n");
         break;

      case 'e':  case 'E': // svuotamento
         stk.svuota();
         printf("Lo stack e' vuoto \n");
         break;

      case 'q':  case 'Q': // uscita
         flg=1;
         break; // uscita
      default: printf("Operazione non consentita \n"); break;
      }
   if (flg==0)
      { printf("Premi un tasto per un'altra operazione\n");
      getch();
      }
   } while (flg!=1);
}
```

20.6 Classe stack realizzata con una lista

Lo stack può ovviamente essere realizzato anche con una lista eliminando gli eventuali limiti della staticità del vettore. Utilizzando una lista e' infatti possibile sfruttare tutto lo spazio disponibile nella memoria *heap* per lo stack. In questo caso la definizione della classe potrebbe essere la seguente.

```
class Stack
{
     int dim;
     Elemento *tos; // puntatore al primo elemento da togliere
     Elemento *bos; // puntatore al primo elemento inserito
public:
     Stack();
     void svuota(void);
     int push(float );
     float * pop (void);
     int isempty(void);
};
```

Nello stack le operazioni di estrazione ed inserimento vengono effettuate dalla stessa parte o in testa o in coda alla lista. Proprio poiché si sta parlando di liste il `tos` in questo caso e' un puntatore a Elemento e non un indice di un vettore.
Le varie operazioni di base che si possono fare sulla lista sono state analizzate in precedenza singolarmente.

20.7 Complessità nella gestione dello *Stack* realizzato con una lista

Quando si va ad effettuare la valutazione della complessità delle operazioni che si possono effettuare su di un certa struttura dati e' necessario fare chiarezza su cosa si intende con le singole operazioni. Nel caso di uno *stack* realizzato con una lista:

- Per inserimento si intende l'operazione di inserzione/push considerando eventuali operazioni di riorganizzazione.
- Per estrazione si intende la sola operazione di eliminazione/pop dell'oggetto considerando le eventuali operazioni di riorganizzazione.

Nella seguente tabella sono riportate le complessità asintotiche, caso migliore, caso peggiore e medio delle varie operazioni che si possono effettuare su uno stack di N oggetti realizzato con una lista concatenata monodirezionale.

In questo caso si possono avere due diverse implementazioni a seconda che si operi una soluzione per la quale si decide di lavorare dalla parte della testa o della coda:

1) inserimento in testa ed estrazione in testa,

2) Inserimento in coda ed estrazione in coda.

Stack con lista modirezionale di N elementi				
Inserimento in coda, estrazione in coda				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Push/inserimento	O(1)	1	1	1
Pop/estrazione	O(N)	N	N	N
Stampa	O(N)	N	N	N
Svuota	O(N)	N	N	N
Isempty	1	1	1	1

Stack con lista modirezionale di N elementi				
Inserimento in testa, estrazione in testa				
Operazione	**Complessità**	**Medio**	**Migliore**	**Peggiore**
Push/Inserimento	O(1)	1	1	1
Pop/Estrazione	O(1)	1	1	1
Stampa	O(N)	N	N	N
Svuota	O(N)	N	N	N
Isempty	1	1	1	1

Da questo si evince che la soluzione migliore per realizzare lo stack con una lista e' organizzare la classe in modo da effettuare le operazioni di *push* e *pop* in testa poiché in tale caso si ha una complessità unitaria per entrambe le operazioni.

21 Bibliografia e Riferimenti

- B. Stroustrup, "Il Linguaggio C++", Addison Wesley, 1995.
- B. W. Kernighan, D. M. Ritchie, "The C Programming Language", Prentice Hall, 1988.
- M. Gori , P. Nesi, E. Pasca, "Pascal e C, guida pratica alla programmazione", Mc Graw-Hill, 1996.
- P. Nesi, "Dispense di fondamenti di informatica", Facoltà di Ingegneria, Università degli Studi di Firenze, 1998.
- S. B. Lippman, J. Lajoie, "C++, Corso di Programmazione", Terza Edizione, Addison Wesley, 2000.
- T.H. Cormen, C. E. Leiserson, R. L. Rivest, "Introduzione agli Algoritmi", Jackson Libri, 1999.