LINGUAGGIO C

AVVERTENZE: I PARAGRAFI DI CUI è PRESENTE SOLAMENTE IL TITOLO NON SONO STATI TRATTATI, O ALMENO COSì MI PARE, QUINDI SE HO SALTATO ARGOMENTI FATTI A LEZIONE AVVISATEMI. NELL'APPENDICE HO INVECE DIRETTAMENTE SALATATO PARAGRAFI DI ARGOMENTI NON FATTI, E PARAGRAFI CHE MI SEMBRAVANO RIDONDANTI CON PARTI DA INSERITE, QUINDI ANCHE IN QUESTO CASO SE DOVREI INCLUDERE ALTRO AVVISATEMI. ERRORI GRAMMATICALI E SINTATTICI SONO PRESENTI SICURAMENTE A BIZZEFFE, FORSE IN FUTURO FARò UN'ALTRA VERSIONE CORREGGENDO GLI ERRORI.

CAPITOLO 1

1.1 Principi Fondamentali

Un programma è costituito da funzioni e da variabili. Le funzioni contengono le **istruzioni** che specificano le operazioni da effettuare Le variabili invece memorizzano i **valori** utilizzati durante l'esecuzione

La funzione main è quella da cui il programma inizia la sua esecuzione, quindi è obbligatoria. Le altre funzioni vengono chiamata all'interno del main. Un metodo per comunicare dati tra funzioni consiste nel fornire alla funzione chiamata una lista di valori, detti argomenti, preparati dalla funzione chiamante. Le parantesi tonde sono utilizzate per racchiudere gli argomenti, mentre le parentesi graffe racchiudono le istruzioni della funzione.

```
printf("Salve, mondo\n");
```

Una funzione viene chiamata con il nome, seguito da una lista di argomenti fra parentesi tonde; perciò quest'istruzione chiama la funzione printf con l'argomento "Salve, mondo\n". printf è una funzione di libreria che stampa un output identificato, in questo caso, dalla stringa di caratteri racchiusa tra apici.

Una sequenza di caratteri fra doppi apici è una stringa di caratteri o stringa costante.

\n → carattere di new line

\t → carattere tabulazione

\b → carattere di backspace

\" → carattere per i doppi apici

\\ → carattere per il backslash

1.2 Variabili ed Espressioni aritmetiche

I commenti sono compresi tra /* e */ vengono ignorati dal compilatore, e sono utilizzati per spiegare il funzionamento del programma.

In C tutte le variabili vanno dichiarate, solitamente prima delle istruzioni eseguibili. Una dichiarazione denuncia le proprietà delle variabili; essa consiste in un nome di tipo ed in una lista di variabili.

Int → variabili di tipo intero (%d)

Float → variabili di tipo decimale (%f)

Char → carattere: un singolo byte (%c)

Short → intero corto

Long → intero lungo (%ld)

Double → numero decimale in doppia precisione

Le istruzioni di assegnamento assegnano alle variabili dei valori iniziali. Le singole istruzioni sono terminate da un punto e virgola.

Il while opera nel seguente modo: viene valutata la condizione tra parentesi. Se è vera, viene eseguito il corpo del ciclo (cioè le istruzioni racchiuse tra parentesi graffe). Quindi la condizione viene ricontrollata e, se è vera, il corpo del ciclo viene rieseguito. Quando la condizione diventa falsa il ciclo termina e l'esecuzione riprende dalla prima istruzione successiva al ciclo.

N.B.

La divisione fra interi effettua un troncamento: qualsiasi parte frazionaria viene scartata. Se voglio moltiplicare una quantità per 5/9 devo fare le operazioni di prodotto e divisione separatamente, altrimenti in programma approssima 5/9 a 0;

Printf è una funzione di uso generale per la stampa di output formattato. Il suo primo argomento è una stringa di caratteri da stampare, nella quale ogni % indica il punto in cui devono essere sostituiti, nell'ordine tutti gli argomenti seguenti; i caratteri immediatamente successivi ad un % indicano la forma nella quale l'argomento dev'essere stampato. Ogni occorrenza del carattere % nel primo argomento è associata al corrispondente argomento. Printf non appartiene al linguaggio C, il quale di per se stesso, non prevede alcuna definizione di funzioni di input / output. Printf è soltanto un'utile funzione presente nella libreria standard delle funzioni normalmente accessibili da programmi C.

Se un operatore aritmetico ha degli operandi interi, l'operazione eseguita è intera. Tuttavia, se un operatore ha un argomento intero ed uno decimale, prima di eseguire l'operazione l'intero viene convertito in un decimale

%d → stampa un intero decimale

%6d → stampa un intero decimale in un campo di almeno sei caratteri

%f → stampa un numero frazionario

%6f → stampa un numero frazionario in un campo di almeno sei caratteri

%.2f → stampa un numero frazionario, con due cifre dopo il punto decimale

%6.2f → stampa un numero frazionario, in un campo di almeno sei caratteri e con almeno due cifre dopo il punto decimale

%o → stampe in ottale

%x → stampe in esadecimale

%s → stamper per le stringhe di caratteri

%% → stampa di %

1.3 **L'istruzione** FOR

N.B.

In ogni contesto in cui è consentito l'uso del valore di una variabile di un certo tipo, è possibile utilizzare anche espressioni di quel tipo ma più complesse.

L'istruzione for è un ciclo, una generalizzazione del while. All'interno delle parentesi tonde ci sono 3 parti separate da punto e virgola. La prima parte l'inizializzazione, viene eseguita una sola volta; la condizione di controllo viene valutata, se vere viene eseguito il corpo del ciclo, altrimenti il ciclo termina; Se il corpo viene eseguito allora viene eseguita la terza parte l'incremento, e la condizione viene nuovamente testata.

1.4 Costanti simboliche

Mantenere, all'interno del codice, numeri espliciti non è una buona abitudine; essi, infatti, non forniscono alcuna informazione ad un eventuale lettore, e risultano difficili da modificare in modo sistematico. Una linea che inizia con #define definisce un nome simbolico, o costante simbolica,

#**define nome testo_da_sostituire**

Tutte le occorrenze di nome (se non siano racchiuse fra apici e non facciano parte di un'altra stringa) vengono rimpiazzate con testo_da_sostituire Per meglio distinguerli dai nomi delle variabili i nomi delle costanti simboliche vengono convenzionalmente scritti a caratteri maiuscoli. Notiamo che, in fondo ad una linea del tipo #define, non viene posto il punto e virgola.

1.5 Input / Output di caratteri

L'input o l'output di un testo, qualsiasi siano la sua sorgente e la destinazione, viene considerato come un flusso di caratteri. Un flusso di testo è una sequenza di caratteri divisi in linee; ogni linea consiste in zero o più caratteri seguiti da un new line.

getchar legge il prossimo carattere di input da un flusso di caratteri e lo restituisce come suo valore.

c=getchar();

la variabile c contiene il prossimo carattere inserito in standard input (solitamente la tastiera)

La funzione putchar stampa un carattere ogni volta che viene invocata:

putchar(c)

stampa, su standard output (solitamente lo schermo) il contenuto della variabile intera c come carattere.

1.5.1 Copia tra i File

EOF (End Of File) è un intero definito in <stdio.h>, ma il suo valore specifico non è significativo, purché sia diverso dal valore di qualsiasi altro char. Usando una costante simbolica, ci assicuriamo che nessuna parte del programma dipenda da questo valore specifico.

1.5.2 Conteggio dei Caratteri

Operatore ++ incremento di uno. Può essere sia prefisso (++i) sia postfissi (i++)

N.B.

Le regole grammaticali del C richiedono che un'istruzione di for possieda sempre un corpo. Il punto e virgola isolato, chiamato istruzione nulla, soddisfa questo requisito.

1.5.3 Conteggio delle Linee

L'istruzione if controlla la condizione fra parentesi e, se questa è vera, esegue l'istruzione (o il gruppo fra parentesi graffe) immediatamente successiva.

L'operatore == è la notazione C per "uguale a"

Un carattere scritto tra singoli apici rappresenta un valore intero, uguale al valore numerico del carattere allo interno del set di caratteri della macchina, che viene detto costante di tipo numerico.

1.5.4 Conteggio delle Parole

|| → operatore logico OR

&& → operatore logico AND

Le espressioni connesse da && e || vengono valutate da sinistra a destra, e la valutazione si blocca non appena viene determinata la verità o la falsità dell'intera espressione.

else specifica un'azione alternativa da intraprendere se la condizione associata ad un if risulta falsa. Una ed una sola delle due istruzioni associate ad un if-else viene eseguita

1.6 Vettori

In C, gli indici di un vettore partono sempre da zero. Un indice può essere una qualsiasi espressione intera e, in quanto tale, può contenere costanti e variabili intere.

Per definizione, i char non sono altro che dei piccoli interi; questo comporta che, nelle espressioni aritmetiche, essi siano identici agli int.

1.7 Funzioni

Una funzione è uno strumento che consente di raggruppare diverse operazioni, il cui risultato può essere riutilizzato in seguito, senza che ci si debba preoccupare di come esso sia stato ottenuto dal punto di vista implementativo. Una definizione di funzione ha il seguente formato:

```
tipo_ritornato nome_funzione (parametro1, parametro2,...)

{

dichiarazioni istruzioni

 }
```

La pria riga dichiara i nomi e di tipi dei parametri, oltre che il tipo del risultato restituito dalla funzione. I nomi usati per i parametri sono locali e non sono visibili e nessun altra funzione.

N.B.

Il termine parametro indica una variabile definita nella lista fra parentesi che compare nella definizione di funzione, mentre il termine argomento sarà il valore usato in una chiamata della funzione stessa. A volte, questo stesso criterio distingue i termini argomento formale ed argomento reale.

L'istruzione return (non è obbligatorio) restituisce al main il valore dell'espressione successiva:

return espressione;

N.B.

Il main è una funzione analoga a qualsiasi altra, anch'esso può restituire un particolare valore al chiamante, che in realtà è l'ambiente nel quale il programma è stato eseguito. Un valore di ritorno nullo indica un corretto completamento; valori diversi da zero segnalano la presenza di condizioni di terminazione eccezionali o scorrette.

```
tipo_ritornato nome_funzione (tipo_parametro1, tipo_ parametro2,...)
```

viene detta prototipo della funzione, deve essere in accordo con la definizione e l'uso della funzione. Il fatto che la definizione o una qualsiasi delle chiamate alla funzione non concordino con questo prototipo costituisce un errore. Questa corrispondenza non è invece richiesta per i nomi dei parametri.

1.8 Argomenti chiamati per valore

In C gli argomenti delle funzioni vengono passati "per valore". Questo significa che i valori degli argomenti vengono forniti alla funzione in variabili temporanee, piuttosto che in quelle d'origine. Da ciò derivano alcune differenze rispetto ai linguaggi che possiedono la "chiamata per riferimento"; in questi casi, infatti, la funzione chiamata accede direttamente all'argomento originale, e non ad una copia locale di esso. La differenza principale consiste nel fatto che, in C, la funzione chiamata non può alterare direttamente una variabile nella funzione chiamante; essa può modificare soltanto la sua copia, privata e temporanea. In caso di necessità, è possibile fare in modo che una funzione modifichi una variabile all'interno della routine chiamante. Il chiamante deve fornire l'indirizzo della variabile (il puntatore), mentre la funzione chiamata deve dichiarare il parametro come un puntatore attraverso il quale accedere, indirettamente, alla variabile stessa. Quando il nome di un vettore è usato come argomento, il valore passato alla funzione è l'indirizzo dell'inizio del vettore stesso: non viene copiato alcun elemento. Indicizzando questo valore, la funzione può usare e modificare qualsiasi elemento del vettore.

1.9 Vettori di caratteri

1.10 Variabili Esterne e Scope

Ogni variabile locale ad una funzione viene realmente creata al momento della chiamata alla funzione stessa, e cessa di esistere quando quest'ultima termina. Per questo motivo, le variabili locali sono dette

anche variabili automatiche. Poiché le variabili automatiche nascono e muoiono con le chiamate alla funzione, esse non possono mantenere il loro valore fra due riferimenti successivi, e devono quindi essere inizializzate opportunamente ad ogni chiamata.

È possibile definire delle variabili esterne a qualsiasi funzione, tramite il loro nome, possono essere utilizzate da ogni funzione del programma. Le variabili esterne sono accessibili globalmente, esse possono sostituire (in parte) le liste di argomenti usate per comunicare i dati da una funzione all'altra. Inoltre queste variabili, non scomparendo al termine dell'esecuzione delle diverse funzioni, sono in grado di conservare il loro valore anche dopo la terminazione della funzione che lo alterato. Una variabile esterna dev'essere definita, una ed una sola volta, al di fuori di qualsiasi funzione; questa operazione implica che venga riservato dello spazio in memoria per questa variabile. La stessa variabile deve anche essere dichiarata in ogni funzione che la utilizza; questa dichiarazione stabilisce il tipo della variabile. La dichiarazione può essere esplicita, con un'istruzione extern, oppure implicita, determinata dal contesto. La dichiarazione con **extern** può essere omessa se la variabile è dichiarata prima della funzione. Se un programma è distribuito in più file sorgenti la prassi consiste nell'inserire le dichiarazioni delle variabili e delle funzioni in un unico file, storicamente chiamato header, incluso da ogni file sorgente con un'istruzione di #include. Il suffisso convenzionale per gli header è .h.

N.B.

"Definizione" si riferisce ai punti nei quali la variabile viene creata o nei quali le viene riservata della memoria; "dichiarazione" si riferisce invece ai punti nei quali viene soltanto stabilita la natura della variabile stessa.

CAPITOLO 2

2.1 Nomi di variabili

I nomi sono costituita da lettere e da cifre, ma il primo carattere deve essere una lettera. Inoltre _ è considerato una lettera ed il linguaggio è case sensitive. Le parole chiave (`if, else, int, ...`) sono riservate e non possono essere usate per i nomi.

2.2 Tipi di Dati e Dimensioni

char → un singolo byte, in grado di rappresentare uno qualsiasi dei caratteri del set locale;
int → un intero, che in genere riflette l'ampiezza degli interi sulla macchina utilizzata;
float → floating-point in singola precisione;
double → floating-point in doppia precisione.

Sono presenti alcuni qualificatori:

- ➔ short almeno di 16 bit ma non più degli int, i long di almeno 32 bit ma non devono essere superati dagli int.
- ➔ signed e unsigned che possono essere associati ai char e int
    - unsigned sono sempre positivi o nulli

A seconda della macchina float, double, long double possono rappresentare diverse ampiezze.

2.3 Costanti

Una costante long è seguita da una 'l' o da una 'L' terminali; un intero troppo grande per essere contenuto in un int verrà considerato long. Le costanti prive di segno sono terminate da una 'u' o una 'U', mentre i suffissi 'ul' ed 'UL' indicano gli unsigned long. Le costanti floating-point contengono il punto decimale, un esponente, oppure entrambi; il loro tipo è sempre double. I suffissi 'f' ed 'F' indicano una costante float, mentre 'l' ed 'L' indicano una costante long double.
Il valore di un intero può essere specificato in decimale, in ottale o in esadecimale. Uno 0 preposto ad un intero indica la notazione ottale; un prefisso 0x (o 0X) indica invece la notazione esadecimale. Anche le costanti ottali ed esadecimali possono essere seguite da un suffisso 'L' che le dichiara di tipo long o da un suffisso 'U' che le dichiara unsigned.
Una costante carattere è un intero, scritto sotto forma di carattere racchiuso tra apici singoli, come 'x'. Il valore di una costante carattere è il valore numerico di quel carattere all'interno del set della macchina. Per

esempio, nel codice ASCII la costante carattere '0' ha valore 48, che non ha nessun legame con il valore numerico 0. Scrivendo '0' invece di 48, che è un valore numerico dipendente dal set di caratteri, il programma risulta indipendente dal valore particolare, oltre che più leggibile. Le costanti carattere possono apparire nelle espressioni numeriche, alla stregua di interi qualsiasi, anche se vengono utilizzate prevalentemente per i confronti con altri caratteri.
Alcuni caratteri, come \n (new line), possono essere rappresentati come costanti o stringhe tramite le sequenze di escape le quali, pur apparendo come stringhe di due caratteri, ne rappresentano uno soltanto.
L'insieme completo delle sequenze di escape è il seguente:

- ➔ \a allarme (campanello)
- ➔ \b backspace
- ➔ \f salto pagina
- ➔ \n new line
- ➔ \r ritorno carrello (return)
- ➔ \t tab orizzontale
- ➔ \v tab verticale
- ➔ \\ backslash
- ➔ \? punto interrogativo
- ➔ \' apice singolo
- ➔ \" doppi apici
- ➔ \ooo numero ottale
- ➔ \xhh numero esadecimale

Un'espressione costante è costituita dal sole costanti e possono essere valutate al momento della compilazione.

Una stringa costante, o costante alfanumerica è una sequenza di zero o più caratteri racchiusa fra doppi apici, che delimitano la stringa. Le stringhe costanti possono essere concatenate al momento di compilazione.

N.B.

'x' è diverso da "x": il primo è un intero, invece il secondo è una stringa di caratteri

Infine esistono le costanti enumerative. Un'enumerazione è una lista di valori interi costanti come:

```
enum boolean {NO, YES};
```

Il primo nome in una costante enumerativa ha valore 0, il secondo 1, e così via, a meno di specificare valori espliciti. Quelli non specificati continuano la progressione

2.4 Dichiarazioni

Tutte le variabili prima di essere utilizzate vanno dichiarate.

Dichiarazione esplicita:

```
int lower, upper, step;

char c, line[1000];
```

Nelle dichiarazioni posso anche inizializzare una variabile se il nome è seguito da = e da un'espressione.

Il qualificatore const può essere applicato alla dichiarazione di qualsiasi variabile, per specificare che il valore non verrà mai alterato

2.5 Operatori Aritmetici

Gli operatori aritmetici binari sono +, -, *, /, e l'operatore modulo %. La divisione intera tronca qualsiasi parte frazionaria. L'operatore % non è applicabile ai float o double. Tutti gli operatori aritmetici sono associativi da sinistra a destra

2.6 Operatori Relazionali e Logici

Gli operatori relazionali sono

> >= < <=

Operatori di uguaglianza == !=

Gli operatori logici sono: && ||

Le espressioni connesse con gli operatori logici sono valutate da sinistra a destra, e la valutazione si blocca non appena si determina la verità (valore 1) o falsità (valore 0) dell'intera espressione.

L'operatore unario di negazione ! converte un operando non nulla in 0, ed un operando nullo in 1.

2.7 Conversioni di Tipo

Quando un operatore ha operandi di tipo diverso, questi vengono convertiti in un tipo comune, secondo un ristretto insieme di regole. In generale, le uniche conversioni automatiche sono quelle che trasformano un operando "più piccolo" in uno "più grande" senza perdita di informazione, come nel caso della conversione di un intero in un floating-point, in espressioni del tipo f+i. Espressioni prive di senso, come per esempio l'uso di un float come indice non sono consentite. Espressioni che possono provocare la perdita di informazioni non sono proibite ma producono un warning. Per quanto concerne la conversione dei caratteri in interi, è necessario fare una sottile osservazione. Il linguaggio non specifica se le variabili di tipo char siano oggetti con o senza segno. Quando un char viene convertito in un int, il risultato può essere negativo. La definizione del C garantisce che nessun carattere che appartiene all'insieme standard di caratteri stampabili della macchina diventi mai negativo.

Le conversioni aritmetiche implicite operano secondo criteri intuitivi. In generale, se un operatore binario come + o * ha operandi di tipo diverso, il tipo "inferiore" viene trasformato nel tipo "superiore" prima di effettuare l'operazione. Il risultato, quindi, appartiene al tipo "superiore". Se non ci sono operandi unsigned, tuttavia, il seguente insieme informale di regole è sufficiente per gestire tutti i casi possibili.

> Se c'è un operando `long double`, l'altro viene convertito in un `long double`.
> In caso contrario, se c'è un operando `double`, l'altro viene convertito in un `double`.
> Altrimenti, se c'è un operando `float`, l'altro viene convertito in un `float`.
> Altrimenti, i `char` e gli `short` vengono convertiti in `int`.
> Infine, se c'è un operando `long`, l'altro viene convertito in un `long`.

Notare che in un'espressione i float non vengono convertiti automaticamente in double.

Anche sugli assegnamenti vengono effettuate delle conversioni; il valore del lato destro viene trasformato nel tipo del valore di sinistra, che è anche il tipo del risultato.

È possibile forzare particolari conversioni, tramite un operatore unario detto cast. Nella costruzione

```
(nome_del_tipo) espressione
```

L'espressione viene convertita nel tipo specificato.

2.8 Operatori di Incremento e Decremento

L'operatore di incremento ++ aggiunge 1 al suo operando, mentre l'operatore di decremento -- sottrae 1. Entrambi possono essere utilizzati in notazione prefissa, la quale incrementa l'operando prima di utilizzarne il valore; e notazione postfissa che incrementa il valore dopo che è stato utilizzato.

2.9 Operatori Bit a Bit

Che io sappia non gli abbiamo fatti, inoltre nel registro non ci sono scritti quindi nada.

2.10 Operatori di Assegnamento ed Espressioni

Quasi tutti gli operatori binari (che hanno un operando dx e sx) hanno un corrispondente operatore di assegnamento del tipo `op=`

+= -= *= /= %= <<= >>= &= ^= |=

```
espr_1 op= espr_2  →  espr_1= (espr_1) op (espr_2)
```

2.11 Espressioni Condizionali

L'espressione condizionale scritta utilizzando l'operatore ternario "?:" fornisce un modo alternativo di scrivere costrutti condizionali:

```
espr_1 ? espr_2 : espr_3
```

viene dapprima valutata l'espressione espr_1. Se essa ha un valore non nullo (se, cioè, risulta vera), allora viene valutata l'espressione espr_2, ed il risultato ottenuto costituisce il valore dell'intera espressione condizionale; in caso contrario, viene valutata espr_3, ed il suo valore è anche quello dell'intero costrutto. Soltanto una, fra le espressioni espr_2 ed espr_3 viene valutata.

```
z=(a>b)?a:b; /* z = max(a,b) */
```

OPERATORI	ASSOCIATIVITÀ
() [] -> .	da sinistra a destra
! ~ ++ -- + - * & (tipo) sizeof	da destra a sinistra
* / %	da sinistra a destra
+ -	da sinistra a destra
<< >>	da sinistra a destra
< <= > >= == !=	da sinistra a destra
&	da sinistra a destra
^	da sinistra a destra
|	da sinistra a destra
&&	da sinistra a destra
||	da sinistra a destra
?:	da destra a sinistra
= += -= *= /= %= &= |= <<= >>=	da destra a sinistra
,	da sinistra a destra

Gli operatori unari +, – e * hanno precedenza maggiore delle rispettive forme binarie.

CAPITOLO 3

STRUTTURE DI CONTROLLO

Le istruzioni di controllo del flusso specificano l'ordine secondo il quale devono essere effettuati i calcoli.

3.1 Istruzioni e Blocchi

In C il punto e virgolo è un terminatore di istruzioni. Le parentesi graffe vengono utilizzate per raggruppare in un'unica istruzione composta, blocco, dichiarazioni e istruzioni, formando un'entità equivalente ad una sola istruzione dal punto di vista sintattico.

3.2 IF – ELSE

Viene utilizzata per esprimere una decisione:

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

L'espressione viene valutata, se risulta vera viene eseguita l'istruzione_1. In caso contrario e se esiste la parte else viene eseguita l'istruzione_2. All'interno di una sequenza di if innestati la mancanza di un else

comporta ambiguità, quindi l'else viene associato automaticamente all'if più interno, a meno di accorgimenti da parte del programmatore (utilizzo delle parentesi graffe)

3.3 ELSE – IF

```
if (espressione_1)
istruzione_1
else if (espressione_2)
istruzione_2
else if (espressione_3)
istruzione_3
else if (espressione_4)
istruzione_4
else
istruzione_5
```

Questa sequenza di istruzioni di if è il modo più generale di realizzare una scelta plurima. Le espressioni vengono valutate nell'ordine in cui si presentano; se una di esse risulta vera, l'istruzione associata viene eseguita e ciò termina l'intera catena.

3.4 SWITCH

L'istruzione `switch` è una struttura di scelta plurima che controlla se un'espressione assume un valore allo interno di un certo insieme di costanti intere, e si comporta di conseguenza.

```
switch (espressione)
{
case espr-cost : istruzioni
                      break
case espr-cost : istruzioni
                      break
default : istruzioni
                 break
}
```

Ogni caso possibile è etichettato da un insieme di costanti intere e di espressioni costanti. Se il valore di espressione coincide con uno di quelli contemplati nei vari casi, l'esecuzione inizia da quel caso particolare. Le espressioni contemplate nei diversi casi devono essere differenti. L'ultimo caso, identificato dall'etichetta `default`, viene eseguito solo se nessuno dei casi precedenti è stato soddisfatto, ed è opzionale. Se esso non compare, e nessuno dei casi elencati si verifica, non viene intrapresa alcuna particolare azione. Le clausole case e default possono occorrere in un ordine qualsiasi. `Break` provoca l'uscita dallo `switch`, infatti l'esecuzione di un'istruzione è seguita dall'esecuzione sequenziale dei casi successivi. L'istruzione `break` può essere impiegata anche per uscire prematuramente da `while, for, do.`

3.5 Cicli – WHILE e FOR

```
while (espressione)
     istruzione
```

Espressione viene valutata; se il suo valore è diverso da zero viene eseguita l'istruzione e l'espressione viene valutata nuovamente. Il ciclo continua fino a quando espressione risulta falsa.

```
for (espr_1; espr_2; espr_3)
     istruzione
```

equivale a:

```
espr_1;
while (espr_2)
{
     istruzione
```

```
        espr_3;
}
```

Ognuna delle componenti del ciclo for può essere tralasciata, anche se i punti e virgola devono essere sempre presenti. L'impiego del for è preferibile quando sono presenti inizializzazioni semplici ed incrementi. Una coppia di espressioni in un costrutto for separate da una virgola "," viene valutata da sinistra a destra ed il tipo ed il valore del risultato coincidono con il valore a destra (la virgola va usata con cautela)

3.6 Cicli – DO – WHILE

Il ciclo do – while controlla, contrariamente agli altri cicli, la condizione d'uscita al termine di ogni iterazione; quindi il corpo del ciclo viene sempre eseguito almeno una volta.

```
do
      istruzione
while (espressione);
```

Viene eseguita l'istruzione, successivamente l'espressione viene valutata; se risulta vera l'istruzione viene eseguita nuovamente; quando diventa falsa il ciclo termina.

3.7 BREAK – CONTINUE

L'istruzione break provoca l'uscita incondizionata da un for, un while oppure un do, nello stesso modo in cui consente l'uscita da uno switch.

L'istruzione continue forza l'inizio dell'iterazione successiva di un for, un while o un do; provoca cioè l'esecuzione immediata della parte di controllo del ciclo.

3.8 GOTO e LABEL

```
goto nome_label;

. . .

nome_label:
```

Porta allo "spaghetti code" quindi viene uno schifo, ma è figo da usare. Comunque manda l'esecuzione del codice al label corrispondente.

CAPITOLO 4

FUNZIONI E STRUTTURA DEI PROGRAMMI

Le funzioni consentono di scomporre problemi complessi in moduli più semplici, sfruttabili anche singolarmente per la risoluzione di problemi diversi. Se strutturate nel modo corretto, le funzioni rendono invisibili al resto del programma i dettagli implementativi che non è necessario esso conosca; esse rendono più chiaro il programma nel suo complesso e ne facilitano notevolmente la manutenzione.

Il C è stato ideato con l'intenzione di rendere le funzioni efficienti e facilmente utilizzabili. Un programma può risiedere in uno o più file sorgente, che possono essere compilati separatamente e caricati insieme unitamente, anche, a funzioni di libreria compilate in precedenza.

4.1 Fondamenti sulle Funzioni

```
Tipo_ritornato nome_funzione(dichiarazioni argomenti)
{
      dichiarazioni ed istruzioni
}
```

Il nome e le parentesi sono le uniche parti fondamentali, infatti se non si specifica il tipo ritornato è int.

La comunicazione tra le funzioni avviene tramite gli argomenti, i valori di ritorno delle funzioni stesse e le variabili esterne. Nel file sorgente posso porre le funzioni in qualsiasi ordine, inoltre il file sorgente può essere diviso in diversi file, contrariamente alla funzione che deve trovarsi in un unico file.

L'istruzione return consente alla funzione di restituire un valore alla funzione chiamante:

```
return espressione;
```

Se necessario l'espressione verrà convertita nel tipo ritornato dalla funzione. Il valore di ritorno nonostante non sia obbligatorio è sempre meglio che venga inserito, lasciando il return senza un'espressione successiva.

4.2 Funzioni Che Ritornano Valori Non Interi

Quando la funzione non restituisce un int è consigliato dichiarare nella funzione chiamante il tipo restituito dalla funzione; infatti in mancanza di un prototipo di una funzione questa viene dichiarata implicitamente dalla sua prima occorrenza in una espressione. Quando una funzione non restituisce valori o non ne richiede è bene usare il tipo void.

4.3 Variabili Esterne

Le variabili esterne sono definite fuori da qualsiasi funzione, e sono perciò a disposizione di più funzioni. Le funzioni stesse sono sempre esterne, perché il C non consente di definire una funzione all'interno di un'altra. Le variabili e le funzioni esterne godono della proprietà che tutti i riferimenti fatti ad esse tramite lo stesso nome, anche se fatti da funzioni compilate separatamente, si riferiscono allo stesso oggetto (lo standard chiama questa proprietà linkaggio esterno). Poiché le variabili esterne sono accessibili globalmente, esse costituiscono un modo alternativo di comunicare dati tra le funzioni, diverso dal passaggio di parametri e dai valori di ritorno. Qualsiasi funzione può accedere ad una variabile esterna attraverso il suo nome, purché tale nome sia stato dichiarato. Se più funzioni devono condividere un elevato numero di variabili, le variabili esterne sono più convenienti rispetto a liste di argomenti molto lunghe (va applicato con cautela, perché troppe connessioni fra i dati delle diverse funzioni, può risultare dannoso per la sua struttura generale)

Le variabili esterne sono utili anche per il loro vasto scope e per la loro "longevità". Infatti, le variabili automatiche sono interne ad una funzione; esse nascono quando viene iniziata l'esecuzione della funzione, e scompaiono quando essa termina. Al contrario, le variabili esterne sono permanenti, quindi mantengono il valore anche fra due chiamate di funzione.

4.4 Regole di Scope

Non mi pare si siano fatti e negli argomenti delle lezioni non li ho trovati quindi non credo, scrivo giusto un due robe. *Forse qualcosa di quello che c'è scritto l'abbiamo fatto, anzi di sicuro*

Lo scope di un nome è la porzione di programma all'interno della quale tale nome può essere usato. Per una variabile automatica, dichiarata all'inizio di una funzione lo scope è la funzione stessa. Variabili locali con lo stesso nome ma dichiarate in funzioni diverse non sono correlate; lo stesso vale per i parametri delle funzioni che non sono altro che variabili locali.

Lo scope di una variabile esterna o di una funzione va dal punto in cui essa è dichiarata al termine del file sorgente in cui si trova.

N.B.

Una dichiarazione rende note le proprietà di una variabile; una definizione provoca anche l'allocazione della memoria riservata a quella variabile.

```
int sp;
double val[MAXVAL];
```

Se appaiono all'esterno di qualsiasi funzione, definiscono le variabili esterne e inoltre servono anche come dichiarazione per il resto del file sorgente.

```
extern int sp;
extern double val[MAXVAL];
```

Invece dichiarano per il resto del file sorgente i nomi e i tipi delle variabili ma non riservano memoria.

Fra tutti i file che costituiscono il programma sorgente, uno solo deve contenere la definizione di una variabile esterna; gli altri possono contenere soltanto dichiarazioni di extern che consente loro di utilizzare la variabile.

4.5 Header File

4.6 Variabili STATIC

La dichiarazione di static applicata ad una variabile esterna o ad una funzione, ne limita lo scope del file sorgente nel quale si trova. Per creare una variabile static basta anteporre ad una normale dichiarazione la parola chiave static.

La dichiarazione static può essere applicata anche a variabili interne; in questo caso consente alla variabile di mantenere il proprio valore anche fra due chiamate successive della funzione alla quale appartiene. Una variabile automatica dichiarata static non scompare al termine dell'esecuzione della funzione ma continua ad esistere anche dopo che la funzione è terminata.

4.7 Variabili REGISTER

Una dichiarazione register avvisa il compilatore che la variabile in questione verrà utilizzata frequentemente. L'idea è che le variabili register debbano essere collocate nei registri della macchina, il cui impiego può consentire di ottenere programmi più brevi e più veloci. I compilatori sono liberi di ignorare tale avvertimento.

```
register int x;

register char c;
```

Essa può essere applicata soltanto alle variabili automatiche ed ai parametri formali di una funzione.

N.B.

Non si può conoscere l'indirizzo di una variabile dichiarata register

4.8 Struttura a Blocchi

La dichiarazione delle variabili (e la loro inizializzazione) può seguire la parentesi graffa sinistra che introduce una qualsiasi istruzione composta, e non soltanto la parentesi che indica l'inizio della funzione. Le variabili dichiarate in questo modo nascondono quelle con nomi uguali dichiarate in blocchi più esterni, e continuano ad esistere fino a che non viene raggiunta la parentesi graffa destra che chiude il blocco.

```
if (n>0)
{
      int i; /* dichiara una nuova variabile i */
      for (i=0; i<n; i++)
      ....
}
```

Lo scope della variabile i è il ramo "then" dell'if; questa i non è correlata ad alcuna i all'esterno del blocco.

Una variabile automatica, dichiarata ed inizializzata in un blocco, viene inizializzata ogni volta che il blocco entra in esecuzione. Una variabile static viene inizializzata soltanto alla prima esecuzione del blocco.

Anche le variabili automatiche ed i parametri formali nascondono le variabili esterne e le funzioni con lo stesso nome.

```
int x;
int y;
f(double x)
{
      double y;
```

```
        ....
}
```

Le occorrenze di x all'interno della funzione f si riferiscono al parametro di tipo double, mentre quelle all'esterno fanno riferimento ad un int. La stessa cosa accade con la variabile y.

Per motivi di chiarezza, è meglio non usare nomi di variabili che nascondono nomi più esterni.

4.9 Inizializzazione

In assenza di un'inizializzazione esplicita, le variabili esterne e quelle static vengono inizializzate a zero. Inoltre l'inizializzatore dev'essere un'espressione costante; l'inizializzazione viene fatta una volta, concettualmente prima dell'inizio dell'esecuzione del programma.

Le variabili automatiche e quelle dichiarate register, invece, hanno valori iniziali indefiniti (sono, cioè, "sporche"). L'inizializzazione viene effettuata ogni volta che inizia l'esecuzione della funzione o del blocco interessato. Per quest'ultimo tipo di variabili, l'inizializzatore può non essere una costante, ma può consistere in una qualsiasi espressione nella quale compaiano valori definiti in precedenza ed anche, eventualmente, chiamate di funzione. Le inizializzazioni delle variabili automatiche non sono altro che un modo più compatto di scrivere degli assegnamenti.

Le variabili scalari possono essere inizializzate al momento della loro definizione, postponendo al loro nome un uguale ed un'espressione.

Un vettore può essere inizializzato postponendo alla sua dichiarazione una lista di valori iniziali, racchiusi tra parentesi graffe e separati da virgole. Se la dimensione è omessa il compilatore calcola la lunghezza in base al numero di elementi presenti nell'inizializzazione. Se gli elementi sono di numero inferiore alla dimensiono gli altri vengono inizializzati a zero se il vettore è una variabile esterna o static; assumono valori indefiniti se il vettore è un variabile automatica.

I vettori di caratteri sono un caso particolare, infatti possono essere inizializzati tramite una stringa costante (racchiusa tra doppi apici).

4.10 Ricorsione

In C, le funzioni possono essere usate in modo ricorsivo; cioè, una funzione può richiamare sé stessa direttamente o indirettamente. Quando una funzione si richiama ricorsivamente, ogni chiamata provoca la creazione di un insieme completo delle variabili automatiche, nuovo ed indipendente dall'insieme precedente. La ricorsione richiede l'impiego di notevoli quantità di memoria, perché è necessario mantenere uno stack dei valori da processare, ed inoltre difficilmente risulta particolarmente veloce. Tuttavia il codice ricorsivo è più compatto. La ricorsione è particolarmente conveniente per la gestione di strutture dati definite ricorsivamente.

4.11 Il Preprocessore C

Il C fornisce alcune particolari funzionalità del linguaggio per mezzo di un preprocessore che, concettualmente, costituisce la prima fase, separata, della compilazione. Le due funzionalità più utilizzate sono #include, per includere, durante una compilazione, i contenuti di un particolare file, e #define, per sostituire ad un identificatore una stringa arbitraria di caratteri.

4.11.1 Inclusione di File

L'inclusione di file facilita la gestione di insiemi complessi di #define e di dichiarazioni. Ogni linea di codice nella forma

```
#include "nome-file"
```

o

```
#include <nome-file>
```

viene sostituita con il contenuto del file nome-file. Se il nome-file è racchiuso tra apici, la ricerca del file inizia dalla directory nella quale si trova il programma sorgente; se il nome-file non viene trovato in questa directory, o è racchiuso tra < > la ricerca prosegue secondo regole dipendenti dall'implementazione. UN file incluso può a sua volta contenere linee di tipo #include.

Solitamente #include ha lo scopo di includere istruzioni #define e dichiarazioni extern comuni, oppure di accedere a prototipi di funzione dichiarati in alcuni header <stdio.h>.

4.11.2 Sostituzione delle Macro

Una definizione ha la forma

```
#define nome testo-da-sostituire
```

Essa richiede una sostituzione di macro del tipo più semplice: tutte le successive occorrenze di nome devono essere sostituite con il testo-da-sostituire. Il nome in una #define ha la stessa forma di un nome di variabile; il testo da sostituire è, invece, arbitrario. Il testo da sostituire occupa la parte restante della linea, ma testi molto lunghi possono proseguire su più linee, purché ognuna di esse, ad eccezione dell'ultima, sia terminata dal carattere \.

Lo scope di un nome definito con una #define va dal punto in cui la #define si trova fino al termine del file sorgente.

Una definizione può sfruttarne altre date in precedenza.

Le sostituzioni vengono effettuate soltanto sui token, e non hanno luogo in caso di stringhe racchiuse fra apici.

```
#define forever for(;;)
```

È anche possibile definire macro con argomenti, in modo che il testo da sostituire dipenda dai parametri delle diverse chiamate:

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

La definizione di un nome può essere annullata con l'istruzione #undef essa, normalmente, viene utilizzata per assicurarsi con una funzione sia definita come tale, piuttosto che come macro.

Se nel testo da sostituire un parametro formale è preceduto da un #, la combinazione viene espansa in una stringa tra apici

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Quando questa macro viene invocata, come in

```
dprint(x/y);
```

essa viene espansa in

```
printf("x/y" " = %g\n", x/y);
```

L'operatore di preprocessor ## consente di concatenare argomenti reali durante l'espansione di una macro.

```
#define paste(front, back) front ## back
```

così

```
paste(name, 1)
```

crea la stringa

```
name1.
```

4.11.3 Inclusione Condizionale

Durante la fase di preprocessing, è possibile controllare il preprocessing stesso attraverso delle istruzioni particolari. Esse, infatti, consentono di inserire segmenti di codice molto selettivo, dipendente dal valore di particolari condizioni, valutate durante la compilazione.

L'istruzione `#if` valuta un'espressione costante intera (che non può includere costanti di tipo `enum`, operatori di cast e `sizeof`). Se l'espressione è diversa da zero, le linee comprese fra `#if` ed il successivo `#endif`, o `#elif` o `#else` vengono incluse

CAPITOLO 5

PUNTAOTORI E VETTORI

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. I puntatori ed i vettori sono strettamente correlati.

5.1 Puntatori e Indirizzi

Una macchina possiede un vettore di celle di memoria numerate o indirizzate in modo consecutivo; queste celle possono essere manipolate singolarmente o a gruppi.

Se c è un char e p un puntatore che punta a c, l'operatore unario & fornisce l'indirizzo di un oggetto, perciò l'istruzione

```
p=&c
```

assegna l'indirizzo di c alla variabile p, e si dice che p "punta a" c.

N.B.

L'operatore & si applica soltanto ad oggetti definiti in memoria, ed esso non può essere applicato alle espressioni, alle costanti o alle variabili register

L'operatore unario * è l'operatore di indirezione o deriferimento: quando viene applicato ad un puntatore, esso accede all'oggetto puntato. Un puntatore è vincolato a puntare ad un particolare tipo di oggetto: ogni puntatore punta ad uno specifico tipo di dati.

5.2 Puntatori ed Argomenti di Funzione

Poiché il C passa alle funzioni gli argomenti per valore, la funzione chiamata non ha un modo diretto per alterare una variabile nella funzione chiamante. Il modo per ottenere questo consiste nel passare alla funzione i puntatori agli oggetti e non il loro valore.

5.3 Puntatori e Vettori

Qualsiasi operazione effettuabile indicizzando un vettore può essere eseguita anche tramite i puntatori.

La dichiarazione

```
int a[10];
```

definisce un vettore a di ampiezza 10, cioè un blocco di dieci oggetti consecutivi, chiamati a[0], …, a[9] allora l'assegnamento

```
pa=&a[0];
```

punterà al primo elemento del vettore, quindi per definizione pa+1 punterà al secondo, pa+i all'i-esimo elemento. Per definizione, il valore di una variabile o di un'espressione di tipo vettore è l'indirizzo dell'elemento zero del vettore stesso. Possiamo dire che un'espressione sotto forma di vettori e indici è equivalente ad una che utilizza puntatori e spiazzamenti (offset).

Esiste però una differenza che deve sempre essere tenuta presente. Un puntatore è una variabile, quindi espressioni come pa=a e pa++ sono legali. Ma il nome di un vettore non è una variabile; costruzioni come a=pa ed a++ sono illegali.

Quando il nome di un vettore viene passato ad una funzione, ciò che viene passato è la posizione dell'elemento iniziale.

5.4 Aritmetica degli indirizzi

Se p è un puntatore a qualche elemento di un vettore, allora p++ incrementa p in modo da farlo puntare all'elemento successivo, mentre p+=i lo fa puntare ad i elementi dopo quello puntato correntemente. Queste ed altre costruzioni simili sono le forme più semplici di aritmetica dei puntatori o degli indirizzi.

`alloc(n)`, restituisce un puntatore p ad n caratteri consecutivi, che possono essere usati dal chiamante di alloc per memorizzare dei caratteri. La seconda routine, `afree(p)`, rilascia la memoria acquisita tramite alloc, in modo che possa essere riutilizzata.

La libreria standard fornisce funzioni analoghe a quelle chiamate `malloc` e `free`.

La costante simbolica `NULL` viene spesso usata al posto dello zero, come nome mnemonico per indicare più chiaramente che questo, per un puntatore, è un valore speciale. La costante simbolica `NULL` è definita in `<stdio.h>`.

In primo luogo i puntatori, in certe circostanze, possono essere confrontati. Se p e q puntano a membri dello stesso vettore, operatori relazionali come ==, !=, <, >=, ecc. lavorano correttamente.

In secondo luogo, abbiamo già osservato che un puntatore ed un intero possono esser sommati o sottratti. La costruzione

```
p+n
```

indica l'indirizzo dell'n-esimo oggetto che segue quello attualmente puntato da p. Questo è vero indipendentemente dal tipo di oggetto a cui punta p; n viene dimensionato in base alla dimensione degli oggetti ai quali p punta, e tale dimensione è determinata dalla dichiarazione di p stesso.

5.5 Puntatori a Caratteri e Funzioni

Quando, in un programma, compare una stringa di caratteri, l'accesso ad essa avviene attraverso un puntatore a carattere, cioè una stringa costante viene acceduta tramite un puntatore al suo primo elemento.

Tra le seguenti definizioni esiste un'importante differenza:

```
char amessage[]=”ora è il momento”; /* un vettore */

char *pmessage=”ora è il momento”; /* un puntatore */
```

`amessage` è un vettore, sufficientemente grande da contenere la sequenza di caratteri e lo '\0' che lo inizializzano. I singoli caratteri all'interno del vettore possono cambiare, ma `amessage` si riferisce sempre alla stessa area di memoria. Al contrario, `pmessage` è un puntatore, inizializzato in modo che punti ad una stringa costante; di conseguenza, esso può essere modificato in modo che punti altrove, ma se tentate di modificare il contenuto della stringa il risultato sarà indefinito.

5.6 Vettori di Puntatori e Puntatori a Puntatori

C'è solo l'esempio quindi niente di utile

5.7 Vettori Multidimensionali

Il C fornisce dei vettori multidimensionali rettangolari, anche se, nella pratica, essi vengono usati molto meno dei vettori di puntatori.

Un vettore multidimensionale si dichiara come:

```
tipo nome_variabile[i][j];
```

dove i,j sono valori definiti.

5.8 Inizializzazione di Vettori di Puntatori

5.9 Puntatori e Vettori Multidimensionali

Date le definizioni

```
int a[10][20];

int *b[10];
```

allora a[3][4] e b[3][4] sono entrambi dei riferimenti ad un singolo int sintatticamente corretti. Tuttavia, a è un vettore bidimensionale: per esso sono state riservate 200 locazioni di ampiezza pari a quella di un int. Per b la definizione alloca soltanto 10 puntatori, che non vengono inizializzati; l'inizializzazione dev'essere fatta esplicitamente, in modo statico oppure con del codice apposito. Assumendo che ogni elemento di b punti ad un vettore di 20 elementi, b stesso occuperà 200 allocazioni di ampiezza pari ad un int, più dieci celle per i puntatori. L'importante vantaggio offerto da un vettori di puntatori consiste nel fatto che esso consente di avere righe di lunghezza variabile. In altre parole, non è detto che ogni elemento di b punti ad un vettore di venti elementi.

5.10 Argomenti alle Linee di Comando

Non mi pare si siano fatti

5.11 Puntatori a Funzioni

In C, una funzione non è, di per se stessa, una variabile; tuttavia, è possibile dichiarare dei puntatori alle funzioni,

5.12 Dichiarazioni Complesse

Nada

CAPITOLO 6

STRUTTURE

Una struttura è una collezione contenente una o più variabili, di uno o più tipi, raggruppate da un nome comune per motivi di maneggevolezza. Soprattutto in programmi di dimensioni notevoli, le strutture aiutano ad organizzare dati complessi, in quanto consentono di trattare come un unico oggetto un insieme di variabili correlate.

Un esempio classico di struttura è quello dello stipendio: un impiegato viene descritto da un insieme di attributi, quali il nome, l'indirizzo, il numero della tessera sanitaria, lo stipendio e così via. A loro volta, alcuni di questi attributi potrebbero essere delle strutture: ogni nome ha diverse componenti, così come avviene per ogni indirizzo o stipendio. Un altro esempio, più tipico del C, proviene dalla grafica: un punto è una coppia di coordinate, un rettangolo è una coppia di punti e così via.

Le strutture, che possono essere copiate ed assegnate, passate alle funzioni e da queste restituite.

6.1 Fondamenti sulle Strutture

La parola chiave `struct` introduce una dichiarazione di struttura, che è una lista di dichiarazioni racchiuse fra parentesi graffe. Un nome opzionale, chiamato identificatore o tag della struttura, può seguire la parola `struct`. Il tag identifica questo tipo di struttura, e può essere utilizzato come abbreviazione per la parte di dichiarazione compresa fra le parentesi.

Le variabili specificate nella struttura sono dette membri. Il membro di una struttura, un tag ed una variabile ordinaria (che, cioè, non appartiene ad alcuna struttura) possono avere lo stesso nome senza che questo crei dei conflitti, poiché il contesto consente sempre di distinguerli. Inoltre, gli stessi nomi dei membri possono ricorrere in strutture diverse, anche se, per chiarezza, è sempre meglio usare nomi uguali soltanto per oggetti strettamente correlati.

Una dichiarazione `struct` definisce un tipo. La parentesi graffa di chiusura che chiude la lista dei membri può essere seguita da una lista di variabili, analogamente a quanto avviene per i tipi fondamentali.

`struct { .... } x, y, z;` è sintatticamente analogo a `int x, y, z;`

Una dichiarazione di struttura non seguita da una lista di variabili non riserva alcun'area di memoria; essa descrive soltanto l'aspetto della struttura.

In un'espressione, un membro di una particolare struttura viene individuato attraverso un costrutto del tipo

```
nome-struttura.membro
```

L'operatore di membro di una struttura "." connette il nome della struttura con quello del membro.

Le strutture possono essere nidificate l'una nell'altra.

6.2 Strutture e Funzioni

Una struttura può essere copiata, assegnata come un unico oggetto, indirizzata tramite l'operatore &, oppure manipolata tramite l'accesso ai suoi membri. La copia e l'assegnamento comprendono anche il passaggio di argomenti alle funzioni e la restituzione di valori dalle funzioni. Le strutture comunque non possono essere confrontate. Una struttura può essere inizializzata con una lista di valori costanti, uno per membro; una struttura automatica, poi, può essere inizializzata anche tramite un assegnamento.

I puntatori alle strutture sono del tutto analoghi a quelli delle variabili ordinarie. La dichiarazione:

```
struct nome-struttura *pp;
```

afferma che pp è un puntatore ad una struttura di tipo struct nome_struttura. I puntatori alle strutture sono usati tanto spesso che si è deciso di fornire una notazione alternativa.

`p->membro-della-struttura` si riferisce al membro nominato.

6.3 Vettori di Strutture

La dichiarazione di struttura

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

dichiara un tipo di struttura, key, definisce un vettore keytab di strutture di questo tipo, e riserva memoria per tali strutture. Ogni elemento del vettore è una struttura. Avremmo anche potuto scrivere:

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

6.4 Puntatori a Strutture

Se p è un puntatore ad una struttura, l'aritmetica su p tiene in considerazione la dimensione della struttura, quindi p++ incrementa p di una quantità tale da farlo puntare alla struttura successiva. Non possiamo comunque assumere che la dimensione di una struttura sia pari alla somma delle dimensioni dei suoi membri. A causa di particolari requisiti di allineamento su oggetti differenti, all'interno di una struttura possono esserci "buchi" privi di nome.

6.5 Strutture Ricorsive

L'albero binario è una struttura formata da un nodo per ogni elemento diverso. Ogni noto oltre a contenere eventuali dati a seconda del contesto, contiene il puntatore al nodo figlio di destra e al nodo figlio di sinistra. Ogni nodo comunque può avere 0,1,2 figli ma non di più.

Si nota facilmente come la struttura di un albero sia ricorsiva, quindi le funzioni che meglio si adatta ad operare su di questo sono funzioni ricorsive.

6.6 Analisi delle Tabelle

6.7 TYPEDEF

Per creare nuovi nomi di tipi di dati, il C fornisce uno strumento chiamato `typedef`. Per esempio, la dichiarazione

```
typedef int Lenght;
```

definisce Lenght come sinonimo di int. La sintassi generale è:

```
typedef tipo nuovo-nome-tipo;
```

È bene sottolineare che una dichiarazione typedef non crea esattamente un nuovo tipo; ma aggiunge un nuovo nome ad un tipo già esistente.

6.8 UNION

6.9 Campi di Bit

CAPITOLO 7

INPUT ED OUTPUT

Le funzionalità di input e output non fanno parte del linguaggio C.

7.1 Input e Output Standard

La libreria implementa un modello molto semplice di input e output riguardante i testi. Un flusso di testo consiste in una sequenza di linee; ogni linea termina con un carattere di new line.

Il più semplice meccanismo di input consiste nel leggere un carattere per volta dallo standard input, normalmente la tastiera, usando `getchar`:

```
int getchar(void);
```

ad ogni chiamata, getchar restituisce il successivo carattere in input.

La funzione

```
int putchar(int)
```

viene utilizzata per l'output: `putchar`(c) invia il carattere c allo standard output, che normalmente è il video.

Anche l'output prodotto da `printf` viene inviato allo standard output. Le chiamate a `putchar` e `printf` possono essere intercalate: l'output appare nell'ordine in cui le chiamate sono state effettuate.

Ogni file sorgente che utilizza una qualsiasi funzione di input / output della libreria standard deve contenere la linea

```
#include <stdio.h>
```

prima di quella in cui la funzione viene usata per la prima volta. Quando il nome dell'header è racchiuso fra < e >, la ricerca dell'header stesso avviene in un insieme standard di directory.

7.2 Output Formattato – PRINTF

La funzione di output printf traduce dei valori interni in carattere. `printf` converte, formatta e stampa sullo standard output i suoi argomenti. La stringa di formato contiene due tipi di oggetti: caratteri ordinari, che vengono semplicemente copiati sullo output, e specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo. Ogni specifica di conversione inizia con un `%` e termina con un carattere di conversione.
Tra `%` ed il carattere di conversione possiamo trovare, nell'ordine:

a) Un segno meno, che specifica l'allineamento a sinistra dell'argomento convertito.

b) Un numero che specifica l'ampiezza minima del campo. L'argomento convertito viene stampato in un campo di ampiezza almeno pari a quella data. Se necessario, vengono lasciati degli spazi bianchi a sinistra (o a destra, se è stato richiesto l'allineamento a sinistra) che consentono di raggiungere l'ampiezza desiderata.

c) Un punto, che separa l'ampiezza del campo dalla precisione.

d) Un numero, la precisione, che specifica il massimo numero di caratteri che devono essere stampati da una stringa, oppure il numero di cifre dopo il punto decimale di un numero floating-point, oppure ancora il numero minimo di cifre di un intero.

e) Una h se l'intero dev'essere stampato come short, oppure l (lettera elle) se dev'essere stampato come long.

Tabella 7.1 *Conversioni base di PRINTF.*

CARATTERE	TIPO DELL'ARGOMENTO; STAMPATO COME
`d, i`	`int`; numero decimale.
`o`	`int`; numero ottale privo di segno (senza zero iniziale).
`x, X`	`int`; numero esadecimale privo di segno (senza `0x` o `0X` iniziale), stampato usando `abcdef` o `ABCDEF` per 10, ..., 15.
`U`	`int`; numero decimale privo di segno.
`c`	`int`; carattere singolo.
`S`	`char *`; stampa caratteri dalla stringa fino al raggiungimento di `'\0'` o della precisione.
`f`	`double`; [-]*m.dddddd*, dove il numero delle *d* è dato dalla precisione (il default è 6).
`e, E`	`double`; [-]*m.ddddde±xx* oppure [-]*m.ddddddE±xx*, dove il numero delle *d* è dato dalla precisione (il default è 6).
`g, G`	`double`; usa `%e` o `%E` se l'esponente è minore di -4 o maggiore o uguale alla precisione; altrimenti usa `%f`. Gli zeri superflui non vengono stampati.
`p`	`void *`; puntatore (rappresentazione dipendente dall'implementazione).
`%`	non converte alcun argomento; stampa un `%`.

Un'ampiezza o precisione può essere specificata come `*`, nel qual caso il valore viene calcolato convertendo l'argomento successivo, che dev'essere un `int`. Per esempio, per stampare al più `max` caratteri di una stringa `s`, possiamo scrivere

```
printf("%.*s", max, s);
```

7.3 Liste di Argomenti di Lunghezza Variabile

7.4 Input Formattato – SCANF

La funzione `scanf` è l'analogo, ma per l'input, di `printf`; `scanf` legge caratteri dallo standard input, li interpreta e memorizza il risultato di queste operazioni negli argomenti successivi (ognuno dei quali dev'essere un puntatore, indicando dove registrare l'input convertito). `scanf` termina quando esaurisce la sua stringa di formato, oppure quando riscontra un'inconsistenza fra lo input e le specifiche di controllo.

In genere, la stringa di formato contiene alcune specifiche di conversione, utilizzate per il controllo della conversione dell'input. Tali specifiche possono contenere:

a) Spazi o caratteri di tabulazione, che vengono ignorati.
b) Caratteri normali (non %), che ci si aspetta corrispondano al successivo carattere non bianco del flusso di input.
c) Specifiche di conversione, costituite dal carattere %, da un * opzionale di soppressione dell'assegnamento, da un numero opzionale che specifica la massima ampiezza del campo, da un h, l o L opzionali indicanti la dimensione dell'elemento, e da un carattere di conversione.

Tabella 7.2 *Conversioni base di SCANF*

CARATTERE	DATI IN INPUT; TIPO DELL'ARGOMENTO
`d`	intero decimale; `int *`.
`i`	intero; `int *`. L'intero può essere in ottale (preceduto da uno `0`) oppure in esadecimale (preceduto da `0x` o `0X`).
`o`	intero ottale (preceduto o meno dallo `0`); `int *`.
`x`	intero esadecimale (preceduto o meno da `0x` o `0X`); `int *`.
`c`	caratteri; `char *`. I prossimi caratteri in input (1 per default) vengono inseriti nella posizione indicata; vengono considerati anche i caratteri di spaziatura; per leggere il prossimo carattere non bianco, usate `%1s`.
`s`	stringa di caratteri (non racchiusa fra apici); `char *`, che punta ad un vettore di caratteri sufficientemente grande da contenere la stringa ed uno `'\0'` di terminazione, che verrà aggiunto automaticamente.
`e, f, g`	numero floating-point con segno, punto decimale ed esponente opzionali; `float *`.
`%`	carattere `%`; non viene effettuato alcun assegnamento.

7.5 Accesso a File
7.6 Gestione degli Errori – STDERRED EXIT
7.7 Input e Output di Linee
7.82 Controllo e Conversione della Classe di un Carattere
7.8.3 UNGETC
7.8.4 Esecuzione di Comandi
7.8.5 Gestione della Memoria

Le funzioni `malloc` e `calloc` allocano dinamicamente blocchi di memoria.

```
void *malloc(size_t n)
```

ritorna un puntatore a `n` byte di memoria non inizializzata, oppure `NULL` se la richiesta non può essere soddisfatta.

```
void *calloc(size_t n, size_t size)
```

ritorna un puntatore ad un'area sufficiente a contenere un vettore di `n` oggetti dell'ampiezza specificata, oppure `NULL` se la richiesta non può essere soddisfatta, inoltre la memoria è inizializzata a zero.
Il puntatore restituito da `malloc` e `calloc` ha l'allineamento corretto per gli oggetti in questione, ma deve essere forzato nel tipo appropriato, come in

```
int *ip;
ip=(int *) calloc(n, sizeof(int));
```

`free(p)` libera lo spazio puntato da `p`, dove `p` è un puntatore ottenuto tramite una precedente chiamata a `malloc` o `calloc`.

7.8.6 Funzioni Matematiche

Esistono più di venti funzioni matematiche, dichiarate in `<math.h>`; l'elenco che segue comprende quelle maggiormente utilizzate. Ognuna di esse ha uno o più argomenti di tipo `double`, e ritorna a sua volta un `double`.

`sin(x)`	seno di *x*, con *x* espresso in radianti
`cos(x)`	coseno di *x*, con *x* espresso in radianti
`atan2(y,x)`	arcotangente di *y/x*, in radianti
`exp(x)`	funzione esponenziale *e^x*
`log(x)`	logaritmo naturale (base *e*) di *x* (*x* > 0)
`log10(x)`	logaritmo comune (base 10) di *x* (*x* > 0)
`pow(x,y)`	*x^y*
`sqrt(x)`	radice quadrata di *x* (*x* >= 0)
`fabs(x)`	valore assoluto di *x*

7.8.7 Generazione di Numeri Casuali

La funzione `rand()` calcola una sequenza di interi pseudo-casuali nell'intervallo 0, `RAND_MAX`, che è definito in `<stdlib.h>`. La funzione `srand(unsigned)` stabilisce il seme per rand.

CAPITOLO 8
L'INTERFACCIA DEL SISTEMA UNIX

8.1 Descrittori di File
8.2 I/O a Basso Livello **–** READ e WRITE
8.3 OPEN, CREATE, CLOSE, UNLINK
8.4 Accesso Casuale **–** LSEEK
8.5 Esempio **– Un'implementazione di FOPEN e GETC**
8.6 Esempio **–** Listing di Directory
8.7 Esempio **–** Un Allocatore di Memoria

APPENDICE A
REFERENCE MANUAL

A2.2 Commenti
I caratteri /* introducono un commento, che termina con i caratteri */. I commenti non possono essere nidificati e non possono comparire all'interno di stringhe.

A2.3 Identificatori
Un identificatore è una sequenza di lettere e cifre. Il primo carattere dev'essere una lettera, fra le quali è compreso il carattere underscore _. Le lettere maiuscole sono considerate diverse da quelle minuscole. Gli identificatori possono avere qualsiasi lunghezza.

A2.4 Parole Chiave
Sono quelle tipo if, for, while, non ve le elenco

A2.5 Costanti
Esistono diverse classi di costanti. Ognuna di esse ha un proprio tipo di dati;

```
costante:
costante-intera
costante-carattere
costante-floating
costante-enumerativa
```

A2.6 Stringhe Letterali

Una stringa letterale, detta anche stringa costante, è una sequenza di caratteri racchiusi fra doppi apici. Una stringa è un "vettore di caratteri"

A4. Significato degli Identificatori

Gli identificatori, o nomi, si riferiscono a diverse classi di oggetti: funzioni; tag delle strutture, delle union e delle enumerazioni; membri di strutture o union; costanti enumerative; nomi di tipi definiti con `typedef` e oggetti. Un oggetto, talvolta chiamato variabile, è una locazione di memoria, e la sua interpretazione dipende da due attributi principali: la sua **classe di memoria** ed il suo **tipo**. La classe di memoria determina la durata della vita della memoria associata all'oggetto identificato; il tipo determina il significato dei valori trovati nell'oggetto identificato. Un nome ha anche uno scope, che è la regione del programma nella quale tale nome è conosciuto, ed un linkaggio, che determina se lo stesso nome, in un altro scope, si riferisce allo stesso oggetto o funzione.

A4.1 Classi di Memoria

Esistono due classi di memoria: automatica e statica. Gli oggetti automatici sono locali ad un blocco, all'uscita del quale vengono abbandonati. Se non comprendono alcuna specificazione della classe, oppure se compare lo specificatore `auto`, le dichiarazioni all'interno di un blocco creano oggetti automatici. Gli oggetti dichiarati `register` sono automatici e, se possibile, vengono allocati nei registri della macchina, che hanno un'elevata velocità di accesso.
Gli oggetti statici possono essere locali ad un blocco od esterni a qualsiasi blocco, ma in entrambi i casi

mantengono il loro valore anche fra l'uscita ed il successivo rientro da funzioni e blocchi. All'interno di un blocco, compreso quello che costituisce il corpo di una funzione, gli oggetti statici sono dichiarati esternamente a qualsiasi blocco, cioè allo stesso livello delle definizioni di funzione, sono sempre statici.

A4.2 Tipi Fondamentali

Gli oggetti dichiarati come caratteri (`char`) sono sufficientemente grandi da potere rappresentare qualsiasi membro del set locale di caratteri.
sono disponibili fino a tre tipi di interi: `short int`, `int` e `long int`. Gli oggetti di tipo `int` hanno la dimensione naturale suggerita dall'architettura della macchina; le altre ampiezze vengono fornite per soddisfare esigenze particolari.
Gli interi privi di segno, sono dichiarati usando la parola chiave `unsigned`.
Anche i tipi floating point in singola (`float`), in doppia (`double`) o in extra (`long double`) precisione possono essere sinonimi, ma ogni elemento di questa lista dev'essere preciso almeno quanto quelli che lo precedono.
Le **enumerazioni** sono di un unico tipo ed hanno valori interi; ad ogni enumerazione è associato un insieme di nomi costanti.
Il tipo `void` specifica un insieme vuoto di valori. Esso viene utilizzato come tipo restituito da funzioni che non generano alcun valore.

A4.3 Tipi Derivati

Oltre ai tipi fondamentali, esiste una classe concettualmente infinita di tipi derivati, costruiti a partire dai tipi fondamentali nei seguenti modi:

- **vettori** di oggetti di un certo tipo;
- **funzioni** che ritornano oggetti di un certo tipo;
- **puntatori** ad oggetti di un certo tipo;
- **strutture** composte da una sequenza di oggetti di tipo diverso;
- **union**, in grado di contenere uno fra diversi oggetti di diverso tipo.

In generale, questi metodi di costruzione degli oggetti possono essere utilizzati ricorsivamente.

A4.4 Qualificatori di Tipo

Dichiarare un oggetto come `const` significa che il suo valore non verrà modificato; dichiararlo `volatile` significa, invece, che esso possiede proprietà particolari, significative per l'ottimizzazione.

A6.3 Interi e Floating

Quando un valore di tipo floating viene convertito in un tipo intero, la parte frazionaria viene scartata; se il valore risultante non può essere rappresentato nel tipo intero voluto, il comportamento è indefinito. In particolare, non è specificato il risultato della conversione di valori floating negativi in tipi interi privi di segno.

A6.4 Tipi Floating

Quando un valore floating viene convertito in un altro floating, di precisione pari o superiore a quella di partenza, il valore resta invariato. Quando la precisione del tipo finale è inferiore a quella di partenza, ed il valore è appresentabile, il risultato può essere sia il primo valore rappresentabile inferiore che quello superiore. Se il risultato non è rappresentabile, il comportamento è indefinito.

A6.5 Conversione Aritmetiche

Molti operatori provocano delle conversioni. L'effetto è quello di ridurre gli operandi ad un tipo comune, che è anche il tipo del risultato.

A6.6 Puntatori ed Interi

Un'espressione di tipo intero può essere sommata o sottratta da un puntatore;
Due puntatori ad oggetti di medesimo tipo, appartenenti allo stesso vettore, possono essere sottratti; il risultato viene convertito in un intero.
Un puntatore può essere convertito in un tipo intero sufficientemente grande da contenerlo;
Un puntatore ad un tipo può essere convertito ad un puntatore ad un altro tipo. Il puntatore risultante può provocare errori di indirizzamento se il puntatore di partenza non si riferisce ad un oggetto correttamente allineato in memoria.
Un puntatore ad una funzione può essere convertito in un puntatore ad una funzione di un altro tipo.
Un puntatore può essere convertito ad un altro puntatore il cui tipo è uguale a meno della presenza o assenza di qualificatori sul tipo dell'oggetto puntato. Se i qualificatori sono presenti, il nuovo puntatore è equivalente al vecchio a meno delle restrizioni ad esso dovute al nuovo qualificatore. In assenza di qualificatori le operazioni sull'oggetto sono governate dai qualificatori presenti nella dichiarazione.

A6.7 VOID

Il valore (inesistente) di un oggetto `void` può essa può essere usata soltanto dove non è richiesto alcun valore.

A6.8 Puntatori a VOID

Qualsiasi puntatore può essere convertito nel tipo `void *`, senza alcuna perdita di informazione. Se il risultato viene riportato al tipo originario, ciò che si ottiene è il puntatore iniziale

A7.2 Espressioni Primarie

Le espressioni primarie sono gli identificatori, le costanti, le stringhe e le espressioni racchiuse fra parentesi.

A7.3 Espressioni Postfisse

Nelle espressioni postfisse, gli operatori si raggruppano da sinistra a destra

A7.3.2 Chiamate di Funzione

Una chiamata di funzione è un'espressione postfissa, detta designatore di funzione, seguita da parentesi tonde che racchiudono una lista (che può anche essere vuota) di espressioni di assegnamento separate da virgole.
Un'espressione passata con una chiamata di funzione è detta **argomento**; il termine **parametro** indica invece un oggetto in input (od il suo identificatore) ricevuto da una definizione di funzione o descritto in una dichiarazione di funzione. Talvolta, per indicare la stessa distinzione, vengono utilizzati rispettivamente i termini "argomento (parametro) attuale" e "argomento (parametro) formale". In preparazione ad una chiamata di funzione, di ogni argomento viene fatta una copia; tutto il passaggio di argomenti avviene infatti per valore. In preparazione ad una chiamata di funzione, di ogni argomento viene fatta una copia; tutto il passaggio di argomenti avviene infatti per valore.
Esistono due modi di dichiarare una funzione. Secondo la sintassi più recente, i tipi dei parametri sono espliciti e fanno parte del tipo della funzione; una simile dichiarazione è detta anche prototipo della funzione.
L'effetto della chiamata è indefinito se il numero di argomenti non concorda con il numero di parametri presenti nella definizione della funzione, o se il tipo di un argomento, dopo le trasformazioni, è diverso da quello del parametro corrispondente.

A7.3.3 Riferimenti a Strutture

Un'espressione postfissa seguita da un punto e da un identificatore è ancora un'espressione postfissa. L'espressione che funge da primo operando dev'essere una struttura o una union, e l'identificatore deve indicarne un membro. Il valore dell'espressione è il membro specificato delle strutture o union, ed il suo tipo è del membro.
Un'espressione postfissa seguita da una freccia (composta dai segni `-` e `>`) e da un identificatore è ancora un'espressione postfissa. L'espressione che funge da primo operando dev'essere un puntatore ad una struttura o ad una union, della quale l'identificatore individua un membro. Il risultato si riferisce al membro specificato della struttura o union puntata dal primo operando, ed il suo tipo è quello del membro.

A7.3.4 Incremento Postfisso

Un'espressione postfissa seguita dall'operatore `++` o `--` è ancora un'espressione postfissa. Il valore della espressione è il valore dell'operando. Dopo averne registrato il valore, l'operando viene incrementato (`++`) o decrementato (`--`) di uno.

A7.4.1 Operatori Incrementali Prefissi

Un'espressione unaria preceduta da `++` o `--` è ancora un'espressione unaria. L'operando viene incrementato (`++`) o decrementato (`--`) di 1. Il valore dell'espressione è il valore dopo l'incremento (o il decremento).

A7.4.2 Operatore di Indirizzamento

L'operatore unario `&` preleva l'indirizzo del suo operando. Il risultato è un puntatore all'oggetto o funzione riferita.

A7.4.7 Operatore di Negazione Logica

L'operando dell'operatore `!` dev'essere di tipo aritmetico o puntatore, ed il risultato è 1 se il valore dell'operando è zero, 0 altrimenti. Il tipo del risultato è `int`.

A7.4.8 Operatore SIZEOF

L'operatore `sizeof` produce il numero di byte richiesti per la memorizzazione di un oggetto del tipo del suo operando. L'operando può essere un'espressione, che non viene valutata, oppure un nome di tipo racchiuso tra parentesi.

A7.5 Cast

Un'espressione unaria preceduta da un nome di tipo racchiuso fra parentesi tonde provoca una conversione del valore dell'espressione nel tipo specificato.

Operatori

È presente tutta una spiegazione sugli operatori +,-,*,/%,>,. . . che non dice niente di nuovo a mio avviso

A7.10 Operatori di Uguaglianza

Gli operatori `==` (uguale a) e `!=` (diverso da) sono analoghi agli operatori relazionali, eccetto che per la loro precedenza, più bassa (quindi, `a<b == c<d` vale 1 ogni volta che `a<b` e `c<d` hanno lo stesso valore di verità).

A7.14 Operatore AND Logico

L'operatore `&&` si raggruppa da sinistra a destra. Esso ritorna 1 se entrambi gli operandi sono diversi da zero, 0 altrimenti. Il primo operando viene valutato, inclusi i suoi effetti collaterali; se esso risulta uguale a zero, il valore dell'intera espressione è 0. Altrimenti, viene valutato l'operando di destra e, se è nullo, il valore dell'intera espressione è 0, altrimenti è 1.
Non è necessario che gli operandi siano dello stesso tipo, purché essi siano tutti di tipo aritmetico o puntatore. Il risultato è di tipo `int`.

A7.15 Operatore OR Logico

L'operatore `||` si raggruppa da sinistra a destra. Esso ritorna 1 se uno dei suoi operandi è diverso da zero, 0 altrimenti. Il primo operando viene valutato, inclusi i suoi effetti collaterali; se esso risulta diverso da zero, il valore dell'intera espressione è 1. Altrimenti, viene valutato l'operando di destra e, se non è nullo, il valore dell'intera espressione è 1, altrimenti è 0.
Non è necessario che gli operandi siano dello stesso tipo, purché essi siano tutti di tipo aritmetico o puntatore. Il risultato è di tipo `int`.

A7.16 Operatore Condizionale

espressione-condizionale:
espressione-OR-logico
espressione-OR-logico `?` *espressione* `:` *espressione-condizionale*
Viene valutata, compresi gli effetti collaterali, la prima espressione; se risulta diversa da 0, il risultato è il valore della seconda espressione, altrimenti quello della terza. Soltanto una delle ultime due espressioni viene valutata.

A7.17 Espressioni di Assegnamento

`= *= /= %= += -= <<= >>= &= ^= |=`

A8. Dichiarazioni

Le dichiarazioni specificano l'interpretazione data ad ogni identificatore; non necessariamente esse riservano memoria per l'oggetto associato all'identificatore. Le dichiarazioni che lo fanno sono chiamate **definizioni**.

A8.2 Specificatori di Tipo

Gli specificatori di tipo sono:

specificatore-tipo:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
specificatore-struttura-o-union
specificatore-enumerativo
nome-typedef

A8.3 Dichiarazioni di Strutture e Union

Una struttura è un oggetto composto da una sequenza di membri di vario tipo.
Una volta dato un nome alla struttura, i tag consentono di creare strutture ricorsive: una struttura può contenere un puntatore ad un'istanza di se stessa.
I nomi dei membri ed i tag non sono in conflitto con alcuna delle altre variabili ordinarie. Uno stesso nome di un membro non può comparire due volte in una stessa struttura o union, ma può essere utilizzato in strutture o union diverse.

A8.4 Enumerazioni

Le enumerazioni sono tipi unici, che assumono valori all'interno di un insieme di costanti chiamate enumeratori. In una lista di enumeratori, gli identificatori sono dichiarati come costanti di tipo `int`, e possono apparire in tutti i punti nei quali possono apparire le costanti. Se non compaiono enumeratori con =, i valori delle costanti corrispondenti iniziano dallo 0 ed aumentano di 1 mano a mano che la dichiarazione viene letta da sinistra a destra. Un enumeratore contenente un = assegna all'identificatore associato il valore specificato; gli identificatori successivi continuano la progressione a partire dal valore assegnato.

A8.6 Significato dei Dichiaratori

Una lista di dichiaratori compare dopo una sequenza di specificatori di tipo e di classe di memoria.

A8.7 Inizializzazione

Quando un oggetto viene dichiarato, il suo dichiaratore-iniziale può specificare un valore iniziale da assegnare all'identificatore dichiarato. L'inizializzatore è preceduto dal segno =, ed è un'espressione oppure una lista di inizializzatori racchiusi fra parentesi graffe.
Un oggetto statico (o i suoi membri, se è una struttura o union) non esplicitamente inizializzato viene inizializzato a 0.
Il valore iniziale di un oggetto automatico non esplicitamente inizializzato è indefinito.
L'inizializzatore di un puntatore o di un oggetto di tipo aritmetico è un'espressione semplice, eventualmente racchiusa fra parentesi. L'espressione viene assegnata all'oggetto.
L'inizializzatore di una struttura è un'espressione dello stesso tipo, oppure una lista di inizializzatori per i suoi membri. Se il numero di inizializzatori è inferiore a quello dei membri della struttura, i membri restanti vengono inizializzati a zero.
L'inizializzatore di un vettore è una lista di inizializzatori per i suoi elementi. Se la dimensione del vettore è sconosciuta, ciò che la determina è il numero degli inizializzatori, ed il tipo del vettore diventa completo. Se il vettore ha ampiezza fissata, il numero degli inizializzatori non deve superare quello degli elementi del vettore stesso; se gli inizializzatori sono meno degli elementi, quelli in eccesso fra questi ultimi vengono inizializzati a zero. Un caso speciale è quello del vettore di caratteri, che può essere inizializzato come una stringa;

A8.9 TYPEDEF

Le dichiarazioni comprendenti lo specificatore di classe di memoria `typedef` non dichiarano alcun oggetto; esse si limitano a definire degli identificatori per particolari tipi di dati. Una dichiarazione `typedef` attribuisce un tipo ad ogni nome compreso nei dichiaratori.

A9. Istruzioni

Ad eccezione di casi specifici le istruzioni vengono eseguite in sequenza. Le istruzioni vengono eseguite per il loro effetto, e non hanno valore proprio.

A9.3 Istruzioni Composte

Poiché diverse istruzioni possono essere usate dove, a livello logico, l'operazione sarebbe unica, il C fornisce l'istruzione composta (detta anche "blocco"). Il corpo di una definizione di funzione è un'istruzione composta.

A9.4 Istruzioni di Selezione

In entrambe le forme dell'istruzione `if` l'espressione, che dev'essere di tipo aritmetico o puntatore, viene valutata, con tutti i suoi effetti collaterali, e confrontata con lo 0: se il suo valore è nullo, la sotto istruzione viene eseguita. Nella seconda forma, la seconda sotto istruzione viene eseguita quando l'espressione risulta nulla. L'ambiguità sull'`else` viene risolta associando ogni `else` all'ultimo `if` che ne è privo, incontrato allo stesso livello di nidificazione del blocco.
Il costrutto `switch` provoca il trasferimento del controllo ad una delle sue istruzioni, scelta in base al valore di una particolare espressione, che dev'essere di tipo intero. Tipicamente, la sotto istruzione controllata da uno `switch` è composta. Qualsiasi istruzione all'interno della sotto istruzione può essere etichettata con una o più label `case`. Due `case` associati allo stesso `switch` non possono contenere. Ad ogni `switch` può venire associata al più una label `default`. Gli `switch` possono essere nidificati; ogni label `case` o `default` è associata allo `switch` più interno che la contiene. Quando l'istruzione `switch` viene eseguita, la sua espressione viene

valutata e confrontata con ogni costante associata alle label `case`. Se una di queste costanti risulta uguale al valore dell'espressione, il controllo passa all'istruzione associata a quella label `case`. Se non esistono label `case` con costanti uguali al valore dell'espressione, ed esiste la label `default`, il controllo passa all'istruzione associata a questa label. Se non esiste neppure la label `default`, non viene eseguita alcuna delle sotto istruzioni dello `switch`.

A9.5 Istruzioni di Iterazione

Le istruzioni di iterazione definiscono dei cicli.
Nelle istruzioni `while` e `do`, la sottoistruzione viene eseguita ripetutamente, fino a quando il valore della espressione rimane diverso da zero; l'espressione dev'essere di tipo aritmetico o puntatore. Con l'istruzione `while` il controllo avviene prima di ogni esecuzione dell'istruzione; con l'istruzione `do`, il controllo avviene invece al termine di ogni esecuzione.
Nell'istruzione `for`, la prima espressione viene valutata una sola volta, e specifica l'inizializzazione del ciclo. Non esistono restrizioni sul tipo di quest'espressione. La seconda espressione dev'essere di tipo aritmetico o puntatore; essa viene valutata prima di ogni iterazione, ed il ciclo termina quando il suo valore è 0. La terza espressione viene valutata dopo ogni iterazione, e specifica la reinizializzazione del ciclo. Non esistono restrizioni sul suo tipo. Gli effetti collaterali di ogni espressione vengono completati immediatamente dopo la sua valutazione. Ognuna delle tre espressioni può venire omessa. Omettere la seconda espressione equivale a controllare una costante non nulla.

A9.6 Istruzioni di Salto

Un'istruzione `continue` può comparire soltanto all'interno di un'istruzione di iterazione. Essa trasferisce il controllo alla parte di continuazione del più interno ciclo di iterazione che la contiene.
Un'istruzione `break` può comparire soltanto in un'istruzione di iterazione o in uno `switch`, e termina l'esecuzione dell'istruzione di iterazione (o `switch`) più interna che la contiene; il controllo passa all'istruzione che segue quella appena terminata.
Una funzione restituisce il controllo dell'esecuzione al chiamante tramite l'istruzione `return`. Quando quest'ultima è seguita da un'espressione, il valore restituito al chiamante è quello dell'espressione stessa.

A10.1 Definizione di Funzioni

Una funzione può restituire un valore di tipo aritmetico, una struttura, una union, un puntatore o un `void`, ma non una funzione o un vettore. In una dichiarazione di funzione, il dichiaratore deve specificare esplicitamente che l'identificatore dichiarato è di tipo funzione.

A11. Scope e Link

Un programma non ha bisogno di essere compilato tutto contemporaneamente: il testo sorgente può essere mantenuto in più file contenenti diverse unità di traduzione, e dalle librerie possono essere caricate routine precompilate.
Ci sono due tipi di scope: il primo è lo **scope lessicale** di un identificatore, che è la regione di testo del programma all'interno della quale le caratteristiche dell'identificatore sono conosciute;
il secondo scope è, invece, quello associato ad oggetti e funzioni con linkaggio esterno, che determina le connessioni tra identificatori residenti in unità di traduzione compilate separatamente.

A11.1 Scope Lessicale

Lo scope lessicale di un oggetto o funzione in una dichiarazione esterna inizia al termine del suo dichiaratore e si estende fino al termine dell'unità di traduzione nella quale l'identificatore compare. Lo scope di un parametro di una definizione comincia all'inizio del blocco che definisce la funzione, e si estende per tutta la funzione; lo scope di un parametro in una dichiarazione di funzione termina alla fine del dichiaratore. Lo scope di un identificatore dichiarato all'inizio di un blocco si estende dalla fine del dichiaratore alla fine del blocco. Lo scope di una label è l'intera funzione nella quale compare. Lo scope di un tag di una struttura, di una union o di un'enumerazione o di una costante enumerativa va dalla sua comparsa nello specificatore di tipo al termine dell'unità di traduzione oppure al termine del blocco

A12. Preprocessing

Un preprocessor effettua sostituzioni delle macro, compilazioni condizionali ed inclusioni di file. Le linee che iniziano con il carattere `#`, eventualmente preceduto da spazi bianchi, comunicano con il preprocessor.

A12.4 Inclusione di File

Una linea di controllo della forma

```
#include <nomefile>
```

provoca la sostituzione di questa linea con l'intero contenuto del file **nomefile**. I caratteri del nome **nomefile**

non devono comprendere > o new line, e l'effetto è indefinito se contengono un carattere qualsiasi fra ", ', \ o /*. Il file specificato viene cercato in una serie di luoghi dipendente dall'implementazione.

```
#include "nomefile"
```

cerca dapprima un file associato al file sorgente originale e, se questa ricerca fallisce, il comportamento diviene identico a quello assunto nella prima forma. L'effetto dell'impiego di ', \ o /* nel nome del file rimane indefinito, ma il carattere > è consentito.

A13. Grammatica
In questo paragrafo si trova una lista di costruzioni sintattiche del C utili per rifrescarsi la memori

APPENDICE B
LIBRERIA STANDARD

B1.2 Output Formattato

Le funzioni `printf` forniscono le conversioni per l'output formattato.
La stringa di formato contiene due tipi di oggetti: caratteri normali, che vengono copiato sullo stream di output, e specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo argomento. Ogni specifica di conversione inizia con il carattere `%` e termina con il carattere di conversione.

B1.3 Input Formattato
Le funzioni `scanf` gestiscono la conversione di input formattato. Una specifica di conversione determina la conversione del successivo elemento in input. Di norma, il risul tato viene memorizzato nella variabile puntata dall'argomento corrispondente

B4. Funzioni Matematiche: MATH.H
Sappi che si sono

B5. Funzioni di Utilità: STDLIB.H
L'header `<stdlib.h>` dichiara funzioni relative alla conversione dei numeri, all'allocazione di memoria e funzionalità simili.

```
int rand(void)
```
`rand` restituisce un intero pseudo-casuale compreso fra 0 e `RAND_MAX`, che vale almeno 32767.

```
void srand(unsigned int seed)
```
`srand` usa `seed` come seme per una nuova sequenza di numeri pseudo-casuali. Il seme iniziali è 1.

```
void *calloc(size_t nobj, size_t size)
```
`calloc` restituisce un puntatore allo spazio per un vettore di `nobj` oggetti di ampiezza `size`, oppu-re `NULL` se la richiesta di allocazione non può essere soddisfatta. Lo spazio è inizializzato ad una serie di zeri.

```
void *malloc(size_t size)
```
`malloc` restituisce un puntatore allo spazio per un oggetto di ampiezza pari a `size`, oppure `NULL` se la richiesta di allocazione non può essere soddisfatta. Lo spazio non è inizializzato.

```
void *realloc(void *p, size_t size)
```
`realloc` modifica, portandola a `size`, l'ampiezza dell'oggetto puntato da `p`. I contenuti restano invariati per uno spazio pari al minimo fra la nuova e la vecchia ampiezza. Se la nuova ampiezza è maggiore della vecchia, il nuovo spazio non è inizializzato. `realloc` ritorna un puntatore alla nuova area, oppure `NULL` se la richiesta non può essere soddisfatta, nel qual caso `*p` rimane invariato.

```
void free(void *p)
```
`free` dealloca lo spazio puntato da `p`; essa non fa niente se `p` è NULL. `p` dev'essere un puntatore ad un'area precedentemente allocata con `calloc`, `malloc` o `realloc`.