

La programmazione strutturata e OOP

1.1 Elementi di base della programmazione strutturata

Un calcolatore può essere visto come un sistema in grado di elaborare i dati che gli sono forniti in ingresso, e produrre in uscita dei risultati. Non è in grado di operare se a esso non vengono fornite precise *istruzioni* per elaborare i dati ricevuti. L'insieme di istruzioni che vengono fornite alla macchina in modo ordinato e sequenziale sono dette **programma**.

Un calcolatore non è in genere una macchina *dedicata*, ovvero capace di svolgere solamente un determinato compito, ma è un sistema che può sviluppare compiti diversi in base alle istruzioni che riceve. Il programma deve essere scritto in un linguaggio che la macchina è in grado di comprendere. Un calcolatore può interpretare soltanto istruzioni scritte in un linguaggio elementare, basato su *codici binari*, detto **linguaggio macchina**. Le istruzioni scritte in linguaggi di programmazione più evoluti non sono direttamente utilizzabili dall'elaboratore, ma occorrono particolari programmi, detti **compilatori**, capaci di effettuare una traslazione dal linguaggio evoluto a quello macchina. A ogni modo, per il *programmatore*, ovvero per colui che scrive il programma, l'azione svolta dal *compilatore* è del tutto trasparente. Il programmatore deve conoscere solamente **il linguaggio di programmazione e le regole sintattiche a esso legate**. Naturalmente la stesura di un programma richiede un dettagliato studio preventivo del problema che si vuole risolvere.

■ Il metodo top-down

Lo studio inizia con un'analisi del problema che tende a individuare le caratteristiche fondamentali dell'applicazione e suddividere questa in strutture più semplici che vengono esaminate e suddivise a loro volta in altre strutture. L'analisi dell'applicazione si riduce così allo studio di semplici problemi. Questo metodo di analisi viene definito *top-down*.

■ Gli algoritmi

È evidente che lo studio di un'applicazione, che dovrà portare poi alla scrittura del programma, deve necessariamente iniziare con la stesura di un **algoritmo**, esplicitando quelle regole che descrivano le azioni che debbono essere svolte per elaborare dati e ottenere risultati congruenti. Non è necessario conoscere le regole del linguaggio di programmazione per sviluppare un algoritmo. Per descrivere con un algoritmo un programma bisogna quindi effettuare un'analisi dell'applicazione con la tecnica di tipo top-down e trovare con quali metodi il problema possa essere risolto.

■ I diagrammi di flusso

La descrizione dell'algoritmo può essere fatta sia servendosi di **diagrammi di flusso** (*flow-chart*), ovvero una serie di simboli grafici (rettangoli, rombi, ecc.) collegati tra loro con delle linee orientate e contenenti al loro interno la descrizione delle varie attività da svolgere, o anche con semplice descrizione verbale dei vari passi dell'algoritmo (*pseudocodice*). Nei diagrammi di flusso

vengono utilizzati fondamentalmente quattro tipi diversi di blocchi elementari, rappresentati nella **figura 1.1**.

figura 1.1



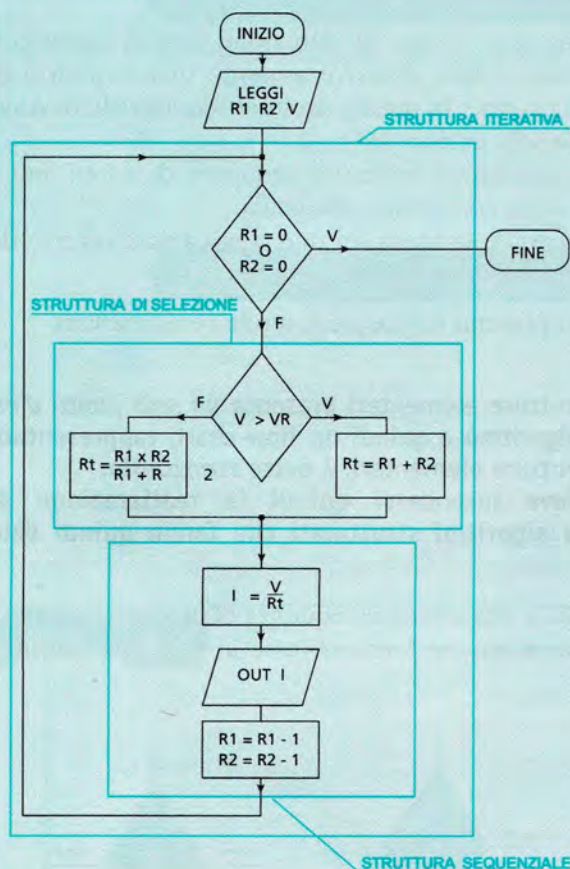
1. Blocco per indicare l'inizio e la fine del programma.
2. Blocco per l'ingresso e l'uscita dei dati.
3. Blocco istruzioni.
4. Blocco di selezione.

Viene dato ora un esempio per strutturare un algoritmo adatto a risolvere il problema sotto esposto.

esercizi

1 Si calcoli la corrente richiesta da un carico resistivo (R_t) costituito da due resistori (R_1 e R_2), che possono essere posti o in parallelo o in serie, alimentati da una tensione continua (V) di assegnato valore. I resistori debbono essere posti in serie se la tensione che alimenta il carico supera un certo valore di riferimento (VR), altrimenti vanno posti in parallelo. Al termine del programma entrambi i valori di R_1 e R_2 vengono decrementati di uno. Il programma, giunto alla fine, inizia un nuovo ciclo e controlla i valori dei resistori. Se uno di essi (o ambedue) raggiunge il valore zero il programma termina.

Dati d'ingresso	Valore resistori	R_1, R_2
	Valore tensione continua	V
Dato di uscita	Corrente assorbita dal carico	I
Costanti assegnate	Tensione di riferimento	VR



Con riferimento all'esercizio viene proposta una rappresentazione dell'algoritmo realizzata sia con un flow-chart che con pseudocodice. L'analisi del problema appena esposto può essere suddivisa nei seguenti passi elementari:

1. introduzione dei dati;
2. verifica che i valori dei resistori siano tali da portare a termine il programma;
3. confronto della tensione d'ingresso con quella di riferimento;
4. realizzazione, a seconda del risultato della verifica precedente, della serie o del parallelo dei resistori;
5. calcolo della corrente sul carico;
6. uscita dei dati;
7. decremento di uno dei valori dei resistori;
8. ritorno al punto 2 per un nuovo ciclo.

Il flow-chart relativo al programma è riportato in **figura 1.2**.

figura 1.2

Costanti, variabili, tipi di dati

Come è possibile verificare dall'algoritmo dell'esercizio illustrato, alcuni dei dati da utilizzare nel programma sono *costanti* (non variano durante lo svolgimento del programma stesso), altri, invece, sono soggetti a variare. I dati del primo tipo sono definiti pertanto *costanti*, mentre i secondi *variabili*. Poiché le *variabili* sono dati soggetti ad assumere valori diversi durante lo svolgimento di un programma, esse vengono associate a un nome simbolico, detto *identificatore*, che può essere considerato come il contenitore del valore vero e proprio della variabile.

Di volta in volta nel contenitore sarà inserito il vero valore che assume la variabile nel corso dello svolgimento del programma. In ogni caso sarà sempre lo stesso identificatore che servirà a individuare la variabile.

Anche per le *costanti* potrà essere utilizzato un nome simbolico per identificarle, ma esse durante lo svolgimento del programma manterranno comunque sempre lo stesso valore. Un altro aspetto che riguarda i dati che vengono utilizzati in un programma è quello relativo al **tipo** che essi rappresentano (per esempio dati numerici interi o con la virgola, dati di tipo letterale, ecc.).

Nell'esempio precedente i valori delle resistenze possono per esempio essere considerati numeri *interi* (se i valori sono introdotti in ohm e non si usano frazioni dell'ohm), il valore di tensione introdotto può essere considerato invece un numero con la virgola (si prendono in considerazione anche i decimi di volt), e così pure la corrente calcolata sarà un numero con la virgola.

Da quanto esposto si può dedurre che il linguaggio di programmazione con cui viene realizzato il programma deve essere in grado di accettare ed elaborare dati di tipo diverso. Si vedrà in seguito, nell'analisi dei vari linguaggi, che in genere essi sono in grado di gestire oltre che dati interi e con virgola, anche altri tipi di dati.

1.2 Strutture

Nella descrizione di ogni algoritmo possono essere sempre individuati solo tre tipi di strutture elementari diverse:

- struttura sequenziale:** costituita da una serie di blocchi istruzione (e/o di ingresso e uscita dati), posti in sequenza l'uno dopo l'altro, descrive appunto una sequenza di azioni che debbono essere svolte in successione. In questa struttura sequenziale devono essere individuati un solo ingresso e una sola uscita.
- struttura iterativa:** ripete per un certo numero di volte una sequenza di azioni fino a quando si verifica o non si verifica una certa condizione assegnata.
- struttura di selezione:** permette di effettuare la scelta tra due o più azioni diverse da svolgere in base al controllo di una condizione assegnata.

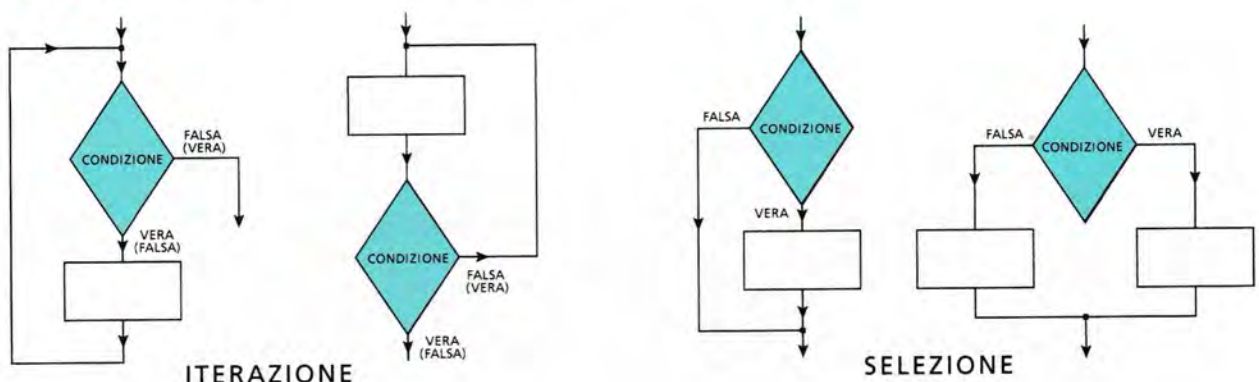
Nel diagramma di flusso di **figura 1.2**, sono presenti tutte e tre le strutture elementari.

Si può osservare che ognuna delle strutture elementari presenta **un solo punto d'ingresso e un solo punto di uscita**. Un algoritmo e quindi un flow-chart, rappresentato solamente con un insieme delle tre strutture elementari, è detto **strutturato**.

Per **programmazione strutturata** deve intendersi quindi la realizzazione di programmi basati esclusivamente su algoritmi strutturati che fanno quindi solo uso delle tre strutture elementari.

Mentre la *struttura sequenziale* è sempre della stessa forma (sequenza di blocchi di azione), le *strutture iterative* e di *selezione* possono assumere forme diverse in base alla modalità con cui vengono svolte le azioni.

figura 1.3



Nella **figura 1.3** sono elencate le varie possibili implementazioni di queste strutture.

All'interno dei blocchi rettangolari possono essere racchiuse altre strutture di tipo sequenziale (o anche iterative e/o di selezione).

► **Configurazioni della struttura iterativa:** come è possibile vedere la struttura iterativa può dare luogo all'implementazione di quattro diverse configurazioni scambiando i valori di vero e falso. Questa struttura darà luogo al *ripetersi delle azioni presenti nel blocco rettangolare* per un certo numero di volte. Si uscirà dalla struttura quando il controllo della condizione restituirà un risultato vero (o falso). È da tenere presente che nella struttura iterativa in cui la condizione viene testata inizialmente il blocco rettangolare può non essere svolto neanche una volta se la condizione risulta falsa (o vera) a seconda dell'implementazione scelta.

Nel secondo caso invece il blocco viene svolto almeno una volta.

► **Configurazioni della struttura di selezione:** sono soltanto due. Questa struttura quindi effettua la scelta o tra due possibili casi o su uno solo.

Come si vedrà nell'Unità dedicata ai linguaggi di programmazione, le strutture elencate si tradurranno in altrettante strutture del linguaggio.

Le precedenti strutture descritte con **pseudocodice** trovano la formulazione in **tabella 1.1**.

tabella 1.1

Iterazione	
<pre> mentre condizione vera o falsa inizio svolgi azione svolgi azione fine </pre>	<pre> ripeti inizio svolgi azione svolgi azione fine finché condizione vera o falsa </pre>
Selezione	
<pre> Se condizione vera inizio svolgi azione svolgi azione fine </pre>	<pre> Se condizione vera inizio svolgi azione fine altrimenti inizio svolgi azione fine </pre>

esempio 1.1

L'algoritmo dell'esercizio 1.1, sviluppato con pseudocodice, assume la seguente forma:

```

leggi i valori di R1, R2 e V
mentre non è ((R1 = 0) o (R2 = 0))
{
  se (V > Vr)
  { Rt = R1 + R2 }
  altrimenti
  { Rt = (R1 * R2) / (R1 + R2) }
  I = V / Rt
  Poni in uscita la corrente I
  Diminuisce di uno R1
  Diminuisce di uno R2
}

```

Le parentesi graffe servono per racchiudere blocchi di codice. Esse sono tipiche del linguaggio C.

In altri linguaggi di programmazione, come per esempio il Pascal, i blocchi di programma sono racchiusi tra le parole *begin* ed *end*. L'asterisco (*) è il simbolo utilizzato per il prodotto e la barra (/) per la divisione.

1.3 Programmazione orientata agli oggetti

Alla base della **programmazione orientata agli oggetti** detta anche **OOP** (*Object Oriented Programming*), che ha attualmente sostituito la più vecchia **programmazione strutturata**, sta il concetto di astrazione dei dati, ovvero la possibilità di creare, da parte dell'utilizzatore di un linguaggio, nuovi tipi di dati più aderenti alla descrizione degli oggetti reali e che offra la possibilità che questi vengano riutilizzati senza la scrittura di nuovo codice.

Un **oggetto** può essere considerato un'applicazione vista come un unico blocco chiuso, contenente sia codice che dati, di cui il programmatore può servirsi nelle proprie applicazioni.

Non interessa al programmatore come l'oggetto sia stato implementato, è essenziale solamente che risponda correttamente all'uso per cui è stato realizzato.

In tutti i linguaggi di programmazione, per i dati che il programma deve utilizzare per l'elaborazione va stabilito a priori il tipo a cui essi appartengono (per esempio, interi, in virgola mobile, caratteri, ecc.).

Similmente nella OOP una **classe** raggruppa oggetti con caratteristiche analoghe. In particolare si può pensare a un oggetto appartenente a una certa classe come un insieme di dati (che costituiscono le **proprietà** possedute dall'oggetto), e dal codice delle funzioni che permettono di accedere ai dati (**metodi**).

Un oggetto viene considerato quindi come una variabile del tipo specificato dalla classe a cui esso appartiene. Si dice che un oggetto è un'**istanza** della classe a cui appartiene. Per creare un oggetto (ovvero fare un'istanza della classe) bisogna fare una dichiarazione usando un identificatore per l'oggetto da creare. Prima di usare un oggetto si deve poi riservare memoria per esso.

Normalmente i dati di un oggetto e il codice che lo implementa sono nascosti al programmatore. Si dice che i dati sono *incapsulati*. Per accedere ai dati di un oggetto si deve ricorrere ai metodi, che sono offerti proprio per interagire con l'oggetto senza poterne modificare il codice.

■ Alcune definizioni

Si danno ora alcune definizioni proprie della programmazione a oggetti.

- **Classe**: si definisce classe il contenitore di un oggetto. La classe fornisce quindi gli attributi necessari per modellare l'oggetto.
- **Creazione di un'istanza**: l'operazione di creazione di un oggetto a partire da una classe viene definita creazione di un'istanza della classe.
- **Variabili dell'istanza**: rappresentano i dati presenti nell'oggetto.
- **Metodi di una classe**: sono le funzioni e le procedure utilizzate in un oggetto di una determinata classe.

Nella programmazione a oggetti debbono essere rispettate tre proprietà fondamentali:

1. **Incapsulamento**: si definisce incapsulamento l'implementazione di dati e codice che costituiscono l'oggetto. Tale implementazione è interna all'oggetto e non deve essere accessibile all'esterno. Normalmente l'accesso a un oggetto deve avvenire solamente attraverso i metodi dell'oggetto.
2. **Ereditarietà**: consiste nella possibilità di creare nuove classi che derivino da classi già esistenti apportando a queste delle modifiche. Le nuove classi ereditano i metodi della classe da cui derivano ma possono aggiungerne anche di nuovi.
3. **Polimorfismo**: è la possibilità che più classi forniscano le stesse proprietà o metodi. Quindi possono essere utilizzati metodi e proprietà di oggetti derivanti da classi che godono di polimorfismo senza conoscere preventivamente a quale classe appartenga l'oggetto.

1.4 Programmazione orientata agli oggetti in Visual Basic

Dei precedenti requisiti della programmazione ad oggetti Visual Basic supporta solamente l'incapsulamento e un particolare metodo di polimorfismo.

Un oggetto in Visual Basic ha:

- **Proprietà:** sono gli attributi che descrivono l'oggetto (ne descrivono per esempio il nome, la forma, le dimensioni, il colore, ecc.).
- **Metodi:** sono le attività che un oggetto può svolgere quando esso viene richiamato da un'applicazione; i metodi possono essere considerati come comandi che VB dà all'oggetto per eseguire determinate azioni.
- **Eventi:** sono azioni che l'oggetto genera in risposta al verificarsi di determinate sollecitazioni.

Per esempio, si esaminino alcune proprietà, metodi ed eventi dell'oggetto **ListBox** (elenco di stringhe racchiuse in una cornice).

- Alcune delle proprietà di questo oggetto sono: le dimensioni, la posizione, il colore dello sfondo, il font dei caratteri, ecc. Come si vede, sono tutti attributi che permettono di definire le caratteristiche dell'oggetto.
- Come metodi dell'oggetto, in altre parole le azioni che il programma in VB può richiedere all'oggetto di eseguire, possono essere prese in considerazione, per esempio, l'aggiunta e la cancellazione di stringhe nell'elenco.
- Gli eventi sono azioni che l'oggetto genera, spesso legate ad attività svolte da parte dell'utente sull'oggetto stesso, come ad esempio la pressione di un tasto o il clic o il doppio clic del mouse. Quando l'azione non è effettuata direttamente sull'oggetto (per esempio clic con il mouse su di esso), ma è indiretta (per esempio pressione di un tasto della tastiera), l'azione viene ricevuta dall'oggetto attivo in quel momento (ovvero, come si dice, l'oggetto che ha il **focus**).

Alla generazione di un evento è associata l'esecuzione di una particolare subroutine il cui codice deve essere scritto dall'utente.

Un oggetto viene identificato con una **variabile oggetto**. Viene definita **interfaccia di un oggetto** l'insieme di proprietà (attributi) e metodi (funzioni) dell'oggetto stesso.

- Nella programmazione in Visual Basic viene spesso utilizzata la seguente espressione:

var_oggetto.tipo_proprietà = xxx

con tale sintassi si indicherà che all'oggetto identificato con la variabile *var_oggetto*, viene applicata la proprietà (*tipo_proprietà*) e a essa si assegna un certo valore (*xxx*).

- Per richiamare un metodo di un oggetto si usa un'espressione simile alla precedente:

var_oggetto.tipo_metodo

dove con *tipo_metodo* si richiama un metodo dell'oggetto indicato.

Elementi del linguaggio C

Per permettere una più facile utilizzazione della programmazione dei microcontrollori PIC usando il linguaggio C, si inseriscono in questa Unità le nozioni di base del linguaggio. Per la stesura e la prova dei programmi presenti nell'unità, si può utilizzare l'ambiente a linea di comando del C++ Builder versione 6 o la nuova versione C++ Builder XE o, infine, uno dei compilatori *free* che possono essere trovati in rete (→ paragrafo 2.12 di questa Unità). Se l'operatore sviluppa i propri programmi seguendo lo standard ANSI (*American National Standards Institute*), il codice sorgente¹ può ritenersi sicuramente *portabile* su macchine diverse da quelle su cui è stato sviluppato.

Gli elementi del **linguaggio C** che saranno trattati seguono il TURBO-C della BORLAND, che rappresenta un'implementazione dello standard ANSI con alcune integrazioni.

Il C, come si vedrà, è sostanzialmente un linguaggio che utilizza *funzioni*, essendo state implementate in esso un numero molto ridotto di parole chiave (trentadue nello standard ANSI e trentanove nel TURBO-C della Borland).

2.1 Tipi di dati

Nelle varie fasi della programmazione per ottenere dei risultati, in genere, vengono elaborati valori di tipo *numerico* o *alfanumerico*. Tali valori sono normalmente raggruppati in modo omogeneo in particolari categorie, dette **tipi di dati**, e rappresentano le varietà di *dati primari*, disponibili per il programmatore, che il linguaggio è in grado di gestire. Per il C essi sono indicati nella **tabella 2.1**.

tabella 2.1

Parola chiave	Tipo	Occupazione memoria	Campo definizione	
char	Carattere	1 byte	0	255
int	Intero	2 byte	-32768	+32767
float	Virgola mobile	4 byte	3.4E-38	3.4E+38
double	Doppia precis.	8 byte	1.7E-308	1.7E+308
void	Indefinito	0 byte	Non definito	

Il tipo *char* può essere impostato inizialmente attraverso il menu di configurazione anche come tipo con segno, occupando in tal caso il campo da -128 a +127.

Nella **tabella 2.1** si è indicato con 3.4E - 38 il valore 3.4×10^{-38} ; e così negli altri casi.

Si fa notare che il campo di definizione dei dati primari e l'occupazione della memoria sono strettamente legati all'architettura del computer e al compilatore con cui si opera. I campi di definizione della **tabella 2.1** sono tipici di un compilatore in ambiente a 16 bit.

¹ Il sorgente è rappresentato dal programma scritto con un editore di testo, in formato ASCII, senza formattazione. Il programma sorgente, per divenire eseguibile, deve essere opportunamente compilato, servendosi del compilatore proprio del linguaggio che si sta adoperando.

Per il C non esiste come dato primario il tipo **stringa** (unione di caratteri alfanumerici). Le stringhe vengono pertanto trattate come un insieme (*array*) di dati tipo `char`. I dati primari possono subire delle modifiche utilizzando i **modificatori di tipo**, costituiti da parole chiave che, preposte ai tipi primari, ne variano l'estensione numerica (**tabella 2.2**).

tabella 2.2

<code>signed</code>	con segno
<code>unsigned</code>	senza segno
<code>long</code>	lungo
<code>short</code>	corto

Non sempre i *modificatori di tipo* operano variazioni sui dati primari su cui agiscono. Nel C si hanno in genere le modifiche esposte nella **tabella 2.3** (i valori sono legati al compilatore utilizzato).

tabella 2.3

Forma abbreviata	Forma completa	da	a	n° byte
<code>unsigned</code>	<code>unsigned int</code>	0	65535	2 byte
<code>unsigned long</code>	<code>unsigned long int</code>	0	4.294.967.295	4 byte
<code>long</code>	<code>long int</code>	-2.147.483.648	+2.147.483.648	4 byte
<code>long double</code>		3.4E-4932	1.1E-4932	10 byte

2.2 Variabili e costanti

■ Variabili

Le grandezze, sia di tipo numerico che di tipo carattere, su cui vengono svolte le elaborazioni necessarie per ottenere i risultati richiesti, se sono soggette, nel corso di un programma, ad assumere valori diversi, sono dette *variabili*.

I dati variabili sono individuati nel C con un nome, detto **identificatore**, che permette di distinguere una variabile dall'altra. Gli identificatori validi possono essere formati da un insieme di caratteri alfabetici e numerici, debbono iniziare necessariamente con un carattere alfabetico e sono significativi (nel caso ne vengano utilizzati di più) solo i primi 32 caratteri.

I caratteri maiuscoli e minuscoli sono considerati diversi: *Alfa*, *alfa* e *ALFA* sono considerati tutti distinti.

Non possono essere usate come identificatori le parole chiave del linguaggio.

Non bisogna mai confondere l'identificatore assegnato a una variabile (*nome*) con il *valore* che essa assume nel corso del programma. Se per esempio si scrive `alfa = 3`, il nome della variabile è *alfa* e il suo valore è 3.

Se poi si scrive `alfa = alfa + 1`, alla variabile *alfa* viene assegnato il nuovo valore 4 (al vecchio valore 3 viene sommato 1).

Nella stesura di un programma deve essere sempre fatta la dichiarazione delle variabili e delle costanti che debbono essere usate nel corso del programma stesso.

Nel C la dichiarazione può essere fatta prima dell'inizio del blocco in cui esse vengono usate. La dichiarazione di più variabili è fatta scrivendo l'elenco degli identificatori, separando ognuno di essi con una virgola e facendo precedere l'elenco dalla parola chiave che rappresenta il tipo di dato (`int`, `float`, `char`, ecc.).

Esempio di dichiarazione di variabili:

```
int alfa, numero;
float somma, risultato, area;
char car, lettera;
int y = 4.
```

Nell'esempio **alfa** e **numero** sono identificatori di variabili a cui verranno assegnati nel corso del programma valori numerici interi, a **somma**, **risultato** e **area** valori numerici con virgola mobile e a **car** e **lettera** dei caratteri. La variabile **y** durante la dichiarazione viene anche inizializzata, ovvero, viene assegnato a essa un valore.

■ Costanti

Per attribuire ad un identificatore un valore che non può essere modificato dopo l'assegnazione, si deve usare il modificatore d'accesso **const**. Si procede come per la dichiarazione delle variabili inizializzate, precedute dalla parola chiave **const**.

Esempi:

```
const float num = 45.765;
const int z = 32;
```

I valori assegnati, in fase di dichiarazione, agli identificatori **num** e **z** non possono essere cambiati nella parte restante del programma.

■ Costanti libere

Si definiscono **costanti (letterali)** quei valori di tipo numerico, letterale o alfanumerico usati durante la scrittura del codice.

Esempi:

```
i = 10
area = 2 × 3.1415 × raggio
```

I numeri 10, 2 e 3.1415 sono delle costanti.

In C sono presenti **costanti intere**, **in virgola mobile**, **a caratteri** ed **enumerative**.

- 1. Costanti intere.** Possono assumere gli stessi valori compresi nel range dei dati del tipo **unsigned int**. A una costante intera è assegnato un tipo di dato primario in base al suo valore. Le costanti intere possono essere **decimali**, **ottali** ed **esadecimali**.

Costanti intere	Base	Modalità di rappresentazione	Esempi
Decimali	10	Senza 0 iniziale	-32576
Ottali	8	Iniziano con 0	03765 (= 2037)
Esadecimali	16	Iniziano con 0x	0x F5A4 (= 62884)

- 2. Costanti in virgola mobile.** Sono rappresentate da numeri con la virgola. Per default sono di tipo **double**.
- 3. Costanti a caratteri.** Sono rappresentate da un carattere racchiuso tra apici: '3', 'A', 'a'. Tra le costanti di caratteri sono da annoverare le così dette *sequenze di Escape* che servono per rappresentare dei caratteri speciali non stampabili. Per introdurre la

sequenza è utilizzato il carattere `\` (barra rovesciata o *back slash*). Nella **tabella 2.4** sono elencate alcune delle sequenze.

Un caso particolare di costanti di caratteri è la **costante stringa**. In questo caso i caratteri sono racchiusi tra doppi apici: "Questa è una stringa".

4. **Costanti enumerative.** Sono costituite da un set ordinato di valori. Nella dichiarazione di una *variabile enumerativa* (fatta con la parola chiave `enum`) sono costanti enumerative i valori racchiusi tra le due parentesi graffe.

tabella 2.4

Caratteri	Tipo
<code>\a</code>	Attivazione segnale acustico
<code>\b</code>	Spazio indietro
<code>\n</code>	Nuova riga
<code>\f</code>	Nuova pagina
<code>\r</code>	Ritorno carrello
<code>\t</code>	Tabulazione orizzontale
<code>\v</code>	Tabulazione verticale
<code>\\</code>	Barra rovesciata
<code>\'</code>	Apice
<code>\"</code>	Doppio apice

Esempio:

```
enum colori {giallo, rosso, verde, azzurro} colore
```

Dove `giallo`, `rosso`, `verde` e `azzurro` sono costanti enumerative e appartengono al nuovo tipo `colori` designato con la variabile `colore`.

Se non diversamente specificato a esse vengono attribuiti i valori costanti interi crescenti 0, 1, 2, 3. È possibile comunque attribuire alle costanti enumerative, durante la dichiarazione, valori diversi da quelli di default.

■ Campo d'azione delle variabili e delle costanti

Un **blocco di programma** è costituito da un insieme di istruzioni racchiuse tra due parentesi graffe. Le variabili (o le costanti) dichiarate all'inizio di un blocco hanno **campo d'azione** in quel blocco e in tutti i sottoblocchi in esso contenuti. Se non sono dichiarate all'inizio del blocco esse hanno campo d'azione dal punto in cui vengono dichiarate fino alla fine del blocco. **Le variabili dichiarate in un blocco sono dette locali. Le variabili dichiarate fuori da tutti i blocchi sono dette globali e hanno un campo d'azione in tutto il programma.**

È possibile dichiarare in un sottoblocco una variabile (o una costante) usando il medesimo identificatore già usato in un blocco più esterno. In questo caso la variabile più interna nasconde quella precedentemente dichiarata, fino al termine del sottoblocco. Quindi la variabile più esterna pur esistendo (è archiviata in memoria), non è visibile.

Per **campo di visibilità** si intende la parte del codice in cui è possibile accedere alle variabili o alle costanti. Nel caso illustrato il campo di visibilità non coincide con il campo d'azione della variabile.

2.3 Schematizzazione dei programmi in C

Nel C per visualizzare dati sul video e per acquisire dati da tastiera si usano due funzioni predefinite `printf()` e `scanf()`.

Tutte le funzioni del C sono in gran parte racchiuse in un **file di libreria** fornito insieme con il compilatore. Sono inoltre forniti all'utente un certo numero di file di **intestazione** (*header*) detti anche **include file** (perché utilizzati con la direttiva `#include`) che contengono i prototipi delle funzioni da usare con i relativi tipi di dati e i nomi delle variabili usate nelle funzioni stesse.

La stesura dei programmi in C deve seguire uno schema ben preciso che deve essere di volta in volta rigorosamente rispettato.

Esso si presenta nel modo seguente:

1. **Elenco dei file include** contenenti i prototipi delle funzioni che debbono essere utilizzate. I nomi dei file debbono essere preceduti dal simbolo `#` e debbono essere racchiusi tra apici o tra i segni `< >`. I file *include* hanno estensione `.h`. Esempio `"stdio.h"` o `<stdio.h>`.
2. **Dichiarazione delle variabili globali** cioè quelle variabili che possono essere utilizzate e modificate da tutte le funzioni presenti nel programma.
3. **Prototipi delle funzioni** cioè una dichiarazione della funzione che indichi sia il tipo del valore che la funzione ritornerà al programma chiamante, sia il tipo di parametri che le saranno passati dal programma chiamante.
4. **Funzione `main()`**; deve essere sempre presente in ogni programma.
5. **Codice di tutte le funzioni utilizzate**, ognuna di esse strutturata come precedentemente esposto. Non si dimentichi che tutte le istruzioni componenti ciascuna funzione debbono essere racchiuse tra due parentesi graffe (blocco di programma).

Si propone un primo esempio di programmazione che permette di esaminare come può essere strutturato un semplice programma in C. Nell'esempio si incontreranno delle parti il cui significato verrà chiarito nel seguito. Il lettore quindi, entrato nell'ambiente del compilatore, scriva fedelmente il testo riportato. I commenti, posti tra i caratteri `/*` e `*/`, servono solo per fornire funzioni esplicative e possono essere omessi. Si faccia attenzione a inserire i punti e virgola `;` dopo le singole istruzioni e le parentesi graffe all'inizio e alla fine del programma. La funzione `printf()` (vedere paragrafo 2.5) serve per visualizzare dati e messaggi sul video, `scanf()` per acquisire dati dalla tastiera. I caratteri `\n` all'interno della funzione `printf()` servono per andare a capo a ogni riga; `%d` permette la visualizzazione di un numero intero, `%f` un numero di tipo *float*, `%c` di un carattere e `%s` di una costante stringa. Notare come per le costanti stringa si è dovuto assegnare (tra parentesi quadre) il numero di caratteri che le compongono più uno, non visibile, che serve per terminare la stringa ed è aggiunto automaticamente dal compilatore. Tutti i programmi in C, come detto, debbono contenere la funzione `main()`.

Il programma scritto può essere messo in esecuzione scegliendo dal menu dei comandi, presente nell'ambiente dell'editore, `RUN` \Rightarrow `RUN`. In ambiente C++ Builder, può essere usato l'apposito pulsante `RUN` (►).

Programma 2.1

```
/* PROGRAMMA DI ESEMPIO */
#include "stdio.h";
int num1;
float num2;
char car = 'A' ;
char stringa1 [7] = "INTERO";
char stringa2 [11] = "TIPO FLOAT";
main()
{
    printf ("\n INTORDUCI UN NUMERO INTERO COMPRESO TRA -32578 E +32767 ");
    scanf ("%d",&num1);
    printf ("\n INTORDUCI UN NUMERO CON VIRGOLA ");
    scanf ("%f",&num2);
    printf ("\n QUESTA E' UNA COSTANTE DI TIPO CHAR: %c",car);
    printf ("\n IL PRIMO NUMERO %d E' %s", num1,stringa1);
    printf ("\n IL SECONDO NUMERO %f E' %s", num2,stringa2);
}
```


2.4 Operatori

Gli operatori servono per manipolare i valori assegnati alle variabili e alle costanti. Di seguito sono elencati gli operatori usati nel C suddivisi in gruppi funzionali:

- ▀ operatore di assegnazione;
- ▀ operatori aritmetici;
- ▀ operatori di relazione;
- ▀ operatori logici;
- ▀ operatori sui singoli bit.

Si tenga presente che, a volte, il medesimo simbolo dell'operatore svolge funzioni diverse in base al contesto in cui è utilizzato.

tabella 2.5

Operatori di assegnazione		
Assegnazione	=	<i>num</i> = 123 assegna alla variabile posta a sinistra del simbolo il valore posto a destra
Assegna il prodotto	*=	<i>num</i> *= <i>a</i> equivale a: <i>num</i> = <i>num</i> * <i>a</i>
Assegna il quoziente	/=	<i>num</i> /= <i>a</i> equivale a: <i>num</i> = <i>num</i> / <i>a</i>
Assegna il resto	%=	<i>num</i> %= <i>a</i> equivale a: <i>num</i> = <i>num</i> % <i>a</i>
Assegna la somma	+=	<i>num</i> += <i>a</i> equivale a: <i>num</i> = <i>num</i> + <i>a</i>
Assegna la differenza	-=	<i>num</i> -= <i>a</i> equivale a: <i>num</i> = <i>num</i> - <i>a</i>
Assegna lo scorrimento a sinistra	<<=	<i>num</i> <<= 1 equivale a: <i>num</i> = <i>num</i> << 1
Assegna lo scorrimento a destra	>>=	<i>num</i> >>= 1 equivale a: <i>num</i> = <i>num</i> >> 1
Assegna l'AND dei bit	&=	<i>num</i> &= <i>a</i> equivale a: <i>num</i> = <i>num</i> & <i>a</i>
Assegna lo XOR dei bit	^=	<i>num</i> ^= <i>a</i> equivale a: <i>num</i> = <i>num</i> ^ <i>a</i>
Assegna l'OR dei bit	=	<i>num</i> = <i>a</i> equivale a: <i>num</i> = <i>num</i> <i>a</i>

tabella 2.6

Operatori aritmetici		
Moltiplicazione	*	<i>num</i> = <i>num1</i> * <i>num2</i> assegna a <i>num</i> il prodotto delle due variabili <i>num1</i> e <i>num2</i>
Divisione	/	<i>num</i> = <i>num1</i> / <i>num2</i> assegna a <i>num</i> il quoziente delle due variabili <i>num1</i> e <i>num2</i>
Addizione	+	<i>num</i> = <i>num1</i> + <i>num2</i> assegna a <i>num</i> la somma delle due variabili <i>num1</i> e <i>num2</i>
Sottrazione	-	<i>num</i> = <i>num1</i> - <i>num2</i> assegna a <i>num</i> la differenza tra le due variabili <i>num1</i> e <i>num2</i>
Più unario	+	<i>num</i> = + <i>num1</i>
Meno unario	-	<i>num</i> = - <i>num1</i>
Resto divisione tra interi	%	<i>res</i> = <i>num1</i> % <i>num2</i> se <i>num1</i> e <i>num2</i> sono due interi, alla variabile <i>res</i> è assegnato il valore del resto della divisione tra <i>num1</i> e <i>num2</i>
Incremento	Preincremento	++ <i>B</i> = ++ <i>A</i> il valore di <i>A</i> viene aumentato di 1 e poi assegnato alla variabile <i>B</i>
	Postincremento	
Decremento	Predecremento	-- <i>B</i> = -- <i>A</i> il valore di <i>A</i> viene diminuito di 1 e poi assegnato alla variabile <i>B</i>
	Postdecremento	

Indichiamo alcuni esempi con gli operatori aritmetici:

<code>int a,b,c,d,g;</code>	<code>/* le variabili a,b,c,d,g sono definite intere */</code>
<code>a = 10;</code> <code>b = 16;</code>	<code>/* si assegnano alle variabili a e b due valori interi */</code>
<code>c = a + b;</code>	<code>/* alla variabile c viene assegnato il risultato della somma tra a e b */</code>
<code>d = c * a;</code>	<code>/* in d finisce il risultato del prodotto tra i valori di c e a */</code>
<code>a = d / b;</code>	<code>/* ad a viene assegnato il valore risultato della divisione tra d e b */</code>
<code>g = d % b;</code>	<code>/* a g viene attribuito il valore del resto della divisione tra d e b */</code>

tabella 2.7

Operatori di relazione		
Maggiore di	<code>></code>	if ($a > b$) se a è maggiore di b
Minore di	<code><</code>	if ($a < b$) se a è minore di b
Uguale a	<code>==</code>	if ($a == b$) se a è uguale a b
Maggiore o uguale	<code>>=</code>	if ($a >= b$) se a è maggiore o uguale a b
Minore o uguale	<code><=</code>	if ($a <= b$) se a è minore o uguale a b
Non uguale	<code>!=</code>	if ($a != b$) se a è diversa da b

tabella 2.8

Operatori logici			
AND	<code>&&</code>	<code>A && B</code>	AND logico tra A e B (A e B di tipo scalare, risultato <i>int</i>).
OR	<code> </code>	<code>A B</code>	OR logico tra A e B (A e B di tipo scalare, risultato <i>int</i>).
NOT	<code>!</code>	<code>!A</code>	negazione logica di A (A di tipo scalare, risultato <i>int</i>).

Nota

Gli scalari sono una categoria di dati che comprende dati aritmetici, enumerativi, puntatori e riferimenti.

A	B	A and B	A or B	A	not A
Falso (0)	Falso (0)	falso (0)	falso (0)	falso (0)	vero (1)
Falso (0)	Vero (1)	falso (0)	vero (1)	vero ($\neq 0$)	falso (0)
Vero ($\neq 0$)	Falso (0)	falso (0)	vero (1)		
Vero ($\neq 0$)	Vero ($\neq 0$)	vero (1)	vero (1)		

In C si ha:

dato vero	<code>$\neq 0$</code>	diverso da zero
dato è falso	<code>$= 0$</code>	uguale a zero

Quando il risultato delle operazioni logiche è **vero**, viene restituito **1** (che è diverso da zero).

Gli operatori finora esaminati agiscono tutti sull'intera variabile. Analizziamo ora gli **operatori che influenzano i singoli bit** della variabile, che deve pertanto essere scritta in binario; non bisogna assolutamente confondere gli operatori logici AND, OR e NOT che operano a livello di variabili con i corrispondenti operatori logici che operano sui singoli bit della variabile.

I primi infatti producono solo un risultato logico di *vero* o *falso*, mentre i secondi operano una vera e propria trasformazione delle variabili.

tabella 2.9

Operatori logici sui singoli BIT			
AND	&	$A \& B$	Opera la AND bit a bit delle variabili (<i>int</i>) A e B
OR		$A B$	Opera la OR bit a bit delle variabili (<i>int</i>) A e B
XOR	^	$A \wedge B$	Opera la XOR bit a bit delle variabili (<i>int</i>) A e B
complemento a uno	~	$\sim A$	Opera il complemento dei singoli bit della variabile (<i>int</i>) A

bit1	bit2	AND (&)	OR ()	XOR (^)	bit	complemento
0	0	0	0	0	0	1
0	1	0	1	1	1	0
1	0	0	1	1		
1	1	1	1	0		

tabella 2.10

Operatori di scorrimento dei BIT		
Scorrimento a destra	>>	$A \gg 1$ tutti bit della variabile (<i>int</i>) A sono traslati di una posizione a destra; l'ultimo bit a destra (meno significativo) viene perso; al posto del primo bit a sinistra (più significativo) viene posto uno zero. Possono effettuarsi più traslazioni variando il numero posto dopo il segno di scorrimento.
Scorrimento a sinistra	<<	$A \ll 1$ tutti bit della variabile (<i>int</i>) A sono traslati di una posizione a sinistra; l'ultimo bit a sinistra (più significativo) viene perso; al posto dell'ultimo bit a destra (meno significativo) viene posto uno zero. Possono effettuarsi più traslazioni variando il numero posto dopo il segno di scorrimento.

Per chiarire la differenza che intercorre tra gli operatori che operano sull'intera variabile e quelli che invece agiscono sui singoli bit, si forniscono alcuni esempi:

<code>int a, b, c, d;</code>	<code>/* si definiscono le variabili a, b, c, d di tipo intero */</code>
<code>a = 5 ; b = 8;</code>	<code>/* si assegnano alle variabili a e b dei valori */</code>
<code>c = a & b</code>	<code>/* si esegue l'operazione logica AND sui singoli bit */</code>
<code>d = a && b</code>	<code>/* si esegue l'operazione logica AND sulle intere variabili */</code>

Dopo l'operazione AND bit a bit, la variabile *c* sarà uguale a:

Valore in binario	
<i>a</i> = 5	0000 0101
<i>b</i> = 8	0000 1000
<i>c</i> = 0	0000 0000

Dopo l'operazione AND sulle variabili *a* e *b*, ricordando che un'operazione logica su due variabili può restituire solo due valori, *vero* (1) o *falso* (0), la variabile *d* sarà uguale:

<i>a</i> = 5	<i>a</i> ≠ 0 quindi	= vero (≠ 0)
<i>b</i> = 8	<i>b</i> ≠ 0 quindi	= vero (≠ 0)
<i>d</i> = 1	vero AND vero	= vero (1)

Deve essere quindi ben chiaro che gli operatori logici sulle intere variabili producono come risultato sempre solo due possibili valori: **0** (falso) o **1** (vero); le operazioni logiche sui bit invece creano valori diversi che dipendono dall'operazione logica effettuata sui singoli bit e dal valore attribuito inizialmente alle variabili.

■ Programma con uso di operatori aritmetici

Programma 2.2

```
#include "stdio.h"
main()
{
    int a,b,c,d,g; /*si definiscono le variabili a, b ,c, d di tipo carattere*/
    a = 10 ;       /*si assegnano alle variabili a e b dei valori*/
    b = 16;
    c = a + b;     /*si esegue la somma tra i valori assegnati ad a e b*/
    d = c * a;     /*si esegue il prodotto tra i valori assegnati ad a e b*/
    printf ("\n a = %d",a); /*si visualizza il valore assegnato ad a*/
    printf ("\n b = %d",b); /*si visualizza il valore assegnato ad b*/
    printf ("\n c = a + b = %d",c); /*si visualizza il valore trovato per c*/
    printf ("\n d = c * a = %d",d); /*si visualizza il valore trovato per d*/
    a = d / b;     /*ad a viene assegnato il nuovo valore
                    risultato della divisione tra d e b*/
    g = d % b;     /*a g viene attribuito il valore del resto
                    della divisione tra d e b*/
    printf ("\n a = d : b = %d",a); /*si visualizza il valore trovato per a*/
    printf ("\n g = resto = %d",g); /*si visualizza il valore di g*/
    getch();      /*attende che si preme un tasto*/
}
```

■ Programma con uso di operatori sui bit

Programma 2.3

```
#include "stdio.h"
main()
{
    char a, b, c, d; /*si definiscono le variabili a, b ,c, d di tipo carattere*/
    a = 5 ; b = 8;    /*si assegnano alle variabili a e b dei valori*/
    c = a & b;        /*si esegue l'operazione logica AND sui singoli bit*/
    d = a && b;        /*si esegue l'operaz. logica AND sulle intere variabili*/
    printf ("\n a = %d",a); /*si visualizza il valore assegnato ad a*/
    printf ("\n b = %d",b); /*si visualizza il valore assegnato ad b*/
    printf ("\n c = a & b = %d",c); /*si visualizza il valore trovato per c*/
    printf ("\n d = a && b = %d",d); /*si visualizza il valore trovato per d*/
    getch();
}
```

2.5 Le funzioni printf() e scanf()

Si prendono ora in esame le due funzioni che permettono di effettuare l'input e l'output verso la console. Queste due funzioni fanno parte della libreria standard del C e i loro prototipi si trovano nel file *include "stdio.h"* (*standard input-output*).

OUTPUT printf()

Sintassi:

printf ("messaggio da visualizzare e direttive di conversione", argomenti).

La parte racchiusa tra virgolette è generalmente costituita da due tipi di elementi diversi:

1. il messaggio da visualizzare, rappresentato da caratteri alfanumerici che vengono visualizzati, così come sono, sul video;
2. le direttive di conversione, che servono invece per rappresentare correttamente gli eventuali argomenti posti dopo la virgola. Le direttive di conversione sono sempre formate dal carattere "%" seguito da una lettera.

Nella **tabella 2.11** sono elencate le direttive di conversione più utilizzate.

tabella 2.11

Direttiva	Viene Visualizzato
%d	Un intero
%c	Un carattere
%u	Un intero senza segno
%f	Un numero in virgola mobile
%e	Un numero in virgola mobile in formato esponenziale
%s	Una stringa
%x	Un intero in formato esadecimale

È da tenere ben presente che ad ogni direttiva deve corrispondere un argomento dello stesso tipo. Gli argomenti non sono altro che gli identificatori delle variabili di cui si vuole visualizzare il contenuto.

Esempio:

```

int num1,num2; float num3;           /*dichiara le variabili di tipo int*/
char alfa,beta;                      /*dichiara le variabili di tipo char*/
num1 = 23; num2 = 55; num3 = 3.14;   /*assegna i valori alle variabili*/
alfa='A'; beta='b';
printf("i numeri sono: %d %d %f", num1, num2, num3);
printf ("\n i caratteri sono: %c %c", alfa, beta);

```

Il frammento di programma dell'esempio visualizza sul video:

```

i numeri sono :    23 55 3.14
i caratteri sono :  A  b

```

Notare che la spaziatura tra il messaggio e tra i singoli caratteri è ottenuta inserendo opportuni spazi sia dopo il messaggio che tra i simboli di formattazione (%d, %f e %c). Vale a dire che il contenuto delle variabili è visualizzato esattamente in corrispondenza della posizione del relativo carattere di formattazione.

Il simbolo speciale “\n” serve per andare a capo. Siccome la barra retroversa “\” precede sempre un carattere speciale e il segno “%” un carattere di formattazione, se in un messaggio si vogliono visualizzare questi simboli bisogna porli duplicati, cioè “\\” visualizza una barra rovesciata e “%%” fa visualizzare il segno di percentuale.

A volte è necessario impostare a priori la lunghezza del campo occupato nella visualizzazione di un dato; in tal caso si impone tale lunghezza inserendo un numero tra il segno “%” e il carattere di formattazione. Esso rappresenterà il numero di spazi riservati al campo. Per esempio “%5d”, riserverà al dato intero da visualizzare 5 posizioni e lo allineerà sulla destra, premettendo spazi bianchi. Se il dato è più lungo dello spazio riservato vengono visualizzate tutte le cifre. Se si vogliono limitare le cifre decimali dopo la virgola, per un numero in virgola mobile, si dovrà porre dopo l'ampiezza del campo un punto seguito dal numero di cifre decimali che si vogliono visualizzare. Per esempio con “%8.3f”, il numero in virgola mobile visualizzato occuperà otto o più spazi (se più lungo), con un totale di tre cifre decimali dopo la virgola.

■ INPUT scanf()

Sintassi:

scanf (“stringa di formato”, indirizzi degli argomenti);

La stringa di formato utilizza per lo più gli stessi formati utilizzati per printf(). Questa volta però la stringa di formato non può contenere messaggi, ma solo caratteri di formato. La novità più rilevante del comando di *input* rispetto a quello di *output* sta nel fatto che **scanf()**, come argomento, richiede non il *valore* di una variabile, ma l'*indirizzo* in cui essa è posta in memoria.

È chiaro che il programmatore non conosce esplicitamente tale indirizzo, ma il compilatore sì; per indicare quindi tale indirizzo si fa precedere il nome della variabile dall'operatore “&”. Per esempio, se la variabile ha nome *alfa*, l'indirizzo di tale variabile sarà: &alfa. Se si scrive la seguente parte di programma:

```

.....
char alfa, beta;
.....
alfa = 'a'; beta = 'B';
.....

```

con le espressioni &alfa e &beta si indicheranno non i valori che hanno le variabili *alfa* e *beta* (cioè rispettivamente “a” e “B”), ma l'indirizzo di memoria dove tali valori saranno memorizzati.

Quindi si può affermare che:

alfa e beta sono gli *identificatori* delle variabili;
 "a" e "B" i *valori* assegnati agli identificatori;
 "&a" e "&B" gli *indirizzi* in cui il compilatore andrà a porre i valori.

Ritornando alla funzione `scanf()`, è possibile dire che, se da *input* si vuole assegnare alla variabile alfa il valore "a", si dovrà scrivere:

```
char car;  
scanf ("%c",&car).
```

Quando viene trovata l'istruzione `scanf()`, il programma si arresta e il computer rimane in attesa che venga introdotto un carattere attraverso la tastiera. Il carattere digitato sarà attribuito all'identificatore `car` e memorizzato all'indirizzo `&car`.

Se si vuole introdurre da tastiera più di una variabile, utilizzando la stessa funzione `scanf()`, queste possono essere separate da uno spazio. Verrà in tal caso usata la seguente sintassi:

```
char car1;  
int num;  
.....  
printf("INTRODUCI UN CARATTERE ED UN INTERO");  
scanf ("%c %d",&car1,&num);
```

In tal caso, lo spazio bianco tra i simboli di formato indica che, in fase di digitazione delle variabili da introdurre, queste devono essere separate da uno o più spazi o da un comando "a capo". Nel caso precedente si dovrà quindi introdurre prima un carattere, poi uno o più spazi bianchi e infine un intero. Il messaggio visualizzato prima dell'acquisizione, avverte l'operatore su cosa deve immettere dalla tastiera.

2.6 Altre funzioni di input e output

A volte, in modo più semplice, possono essere usate altre funzioni di input già predefinite. Tra le principali si indicano:

tabella 2.12

<code>getch()</code>	Acquisisce un carattere da tastiera senza visualizzarlo sul video; non attende che venga digitato il ritorno a capo.
<code>getche()</code>	Acquisisce un carattere da tastiera e lo visualizza su video; non attende che venga digitato il ritorno a capo.
<code>getchar()</code>	Acquisisce un carattere da tastiera e lo visualizza su video; attende che venga digitato il ritorno a capo.
<code>gets()</code>	Acquisisce una stringa da tastiera; la stringa deve terminare con un ritorno a capo.
<code>putchar()</code>	Visualizza un carattere sul video.
<code>puts()</code>	Visualizza una stringa sul video.

La sintassi per utilizzare le precedenti funzioni è illustrata con un esempio.

```
char car1, car2, car3, string[20]; /*definizione di dati tipo char*/  
.....  
car1 = getch(); /*car1 contiene il carattere acquisito da tastiera*/  
car2 = getche(); /*car2 contiene il carattere acquisito da tastiera*/  
car3 = getchar(); /*car3 contiene il carattere acquisito da tastiera*/  
putchar(car1); /*viene visualizzato il carattere car1*/  
gets(string); /*string contiene la stringa acquisita da tastiera*/  
puts(string); /*viene visualizzata la stringa string*/
```


2.7 Le funzioni nel linguaggio C

In C sono presenti solo un numero ridotto di parole chiave; tutte gli altri compiti che il programma deve svolgere sono quindi affidati a delle funzioni.

Quando in un programma è necessario, per esempio, elaborare in modo ripetitivo dei dati, che di volta in volta assumono valori diversi, effettuando su di essi le stesse operazioni, si ricorre alle funzioni. In C una funzione può, svolta i compiti per cui è stata creata, ritornare al programma chiamante un solo valore. Altre volte la funzione svolge particolari compiti senza ritornare alcun valore. In questo caso si dice che la funzione è di tipo **void**.

Una funzione può essere realizzata per mezzo dei seguenti elementi.

- **Prototipo della funzione:** deve essere posto nel codice prima che avvenga la chiamata della funzione.
- **Definizione della funzione:** è la parte del codice che svolge i compiti assegnati alla funzione.

tabella 2.13

Prototipo della funzione

È una dichiarazione della funzione che comprende il *tipo di dato* che la funzione deve ritornare al programma chiamante, il *nome della funzione*, una *lista del tipo di dati* che rappresentano i tipi di dati che, quando viene chiamata, saranno passati a essa.

La funzione elaborerà i dati e fornirà eventualmente un valore, risultato dell'elaborazione, al programma chiamante.

Struttura del prototipo

tipo_funzione nome_funzione (tipo_parametri_formali)

Esempi di prototipi di funzione

Float somma (float, float)	Funzione che restituisce un <i>float</i> e che riceve come parametri due <i>float</i>
Int disegna (int, int, float)	Funzione che restituisce un <i>int</i> e che riceve come parametri due <i>int</i> e un <i>float</i>
Void linea (int, int, int, int)	Funzione che non restituisce alcun valore e riceve come parametri quattro <i>int</i>
Int calcola (void)	Funzione che restituisce un <i>int</i> a cui non vengono passati dati

Definizione della funzione

Comprende il codice che svolge i compiti per cui è stata creata la funzione. È strutturato con il *tipo di dato* che la funzione deve ritornare al programma chiamante, il *nome della funzione*, una *lista di dati* con il relativo *tipo* e, infine, un codice della funzione racchiuso tra due parentesi graffe. Al termine della funzione (prima della chiusura della parentesi graffa) può essere inserita la parola chiave *return* seguita dall'identificatore della variabile da ritornare. Dopo la prima parentesi graffa possono essere dichiarate delle variabili (variabili locali, visibili solo all'interno della funzione).

Struttura del corpo

tipo_funzione nome_funzione (tipo dato1, tipo dato2, ...)

```
{
    eventuale dichiarazione delle variabili
    codice
    return identificatore
}
```

Esempio di funzione

```
Float somma (float x, float y)
{
    int z;
    float a, b;
    codice corpo funzione
    return a;
}
```

Il prototipo della funzione serve a far conoscere preventivamente al compilatore quali tipi di parametri saranno passati alla funzione stessa e quale tipo di dato essa ritornerà al programma chiamante una volta svolto il proprio compito. Nel corso della compilazione sarà inoltre effettuato un severo controllo sul tipo di dati, cosa che renderà più difficile commettere errori.

Nella *definizione della funzione*, a differenza di quanto accadeva nella *dichiarazione del prototipo*, tra le parentesi tonde compaiono sia i tipi dei parametri che i rispettivi identificatori utilizzati.

Ogni programma in C deve sempre contenere almeno una funzione principale detta `main()`. Ad essa possono poi, secondo le necessità, essere aggiunte altre funzioni. Pertanto si può così rappresentare tale funzione:

```
main( )
{
    dichiarazioni variabili locali
    istruzioni con eventuali chiamate ad altre funzioni
}
```

I parametri formali che si forniscono a una funzione possono essere *passati per valore* o *per riferimento*.

- **Passaggio per valore:** nel momento in cui viene chiamata una funzione si crea una copia dei parametri e la funzione opera sulla copia senza possibilità di modificare i valori delle variabili originali.
- **Passaggio per riferimento:** la funzione lavora sulle variabili stesse (facendo riferimento al loro indirizzo), sussiste quindi la possibilità di modifica dei valori assegnati originariamente alle variabili. Nella dichiarazione delle variabili per riferimento, nella dichiarazione della funzione, il tipo di parametro viene fatto seguire dal carattere & (esempio `float& x, int& y`).

Le variabili utilizzate nei programmi possono essere *globali* o *locali* (→ *Campo d'azione delle variabili e delle costanti*, paragrafo 2.2).

- **Variabili globali:** sono quelle variabili che vengono dichiarate all'inizio del programma, al di fuori di qualsiasi funzione. Esse sono *visibili* a tutte le funzioni presenti nel programma, cioè posso essere utilizzate e quindi modificate da tutte le funzioni. Il loro uso non sempre è raccomandabile poiché è facile che siano modificate accidentalmente, anche se, la loro utilizzazione, permette la stesura di programmi più semplici senza il passaggio di parametri alle funzioni.
- **Variabili locali:** tali variabili vengono dichiarate all'interno di ciascuna funzione e, pertanto, sono utilizzabili solo dalla funzione stessa. Le variabili locali non conservano il loro contenuto se avvengono due successive chiamate alla stessa funzione a meno che non vengano referenziate durante la loro dichiarazione con la parola chiave **static**. Se all'interno di una funzione viene dichiarata una variabile con lo stesso nome di una variabile globale, questa viene coperta dalla variabile locale.

Si descrive ora un programma semplice e completo che *chiederà di introdurre da tastiera due numeri di tipo float, li dividerà e mostrerà il risultato della divisione*.

Esso verrà impostato utilizzando un programma principale `main()`, che chiederà di introdurre i numeri e chiamerà poi la funzione `dividi()`; tale funzione a sua volta eseguirà la divisione e, di nuovo poi, cederà il controllo al programma principale che visualizzerà sul video il risultato ottenuto.

Il programma sarà realizzato in tre varianti diverse:

- la prima, che utilizzerà variabili globali e non farà uso del passaggio dei parametri;
- la seconda, che, invece, farà uso del passaggio di parametri e ritornerà il valore del risultato alla funzione chiamante;
- la terza, infine, che userà una variabile globale per il risultato ed effettuerà il passaggio dei parametri alla funzione chiamata.

■ Programma con uso di funzione senza passaggio di parametri

Programma 2.4

```
/*programma che esegue la divisione tra due float. Utilizza variabili
   globali senza passaggio di parametri alla funzione chiamata*/
#include "stdio.h"
float num1,num2,risul;      /*dichiarazioni delle variabili globali*/
void dividi();              /*prototipo della funzione senza parametri*/
main(){                     /*inizio del programma principale*/
printf ("INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE ");
scanf ("%f",&num1);         /*attende che da tastiera venga digitato il
                               numero e lo assegna alla variabile num1*/
printf ("INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE "); /*richiede secondo
numero*/
scanf ("%f",&num2);         /*preleva il secondo numero assegnandolo a
                               num2*/
dividi();                   /*chiama la funzione per la divisione*/
printf ("\n IL RISULTATO DELLA DIVISIONE E' %f ",risul);
getch();
    }                       /*fine del programma principale*/

void dividi() {              /*funzione che esegue la divisione*/
    risul = num1 / num2; /*viene eseguita la divisione e il risultato
                           è assegnato alla variabile globale risul*/
    }                       /*fine della funzione*/
```

Osservazioni

- La funzione `dividi` è stata dichiarata *void* in quanto non deve ritornare alcun valore perché modifica direttamente la variabile globale *risul*.
- Tutte le istruzioni debbono essere chiuse da un *punto e virgola* (;).
- Il carattere speciale `\n` all'interno della funzione `printf()` serve per andare a capo.

■ Programma con uso di funzione con passaggio di parametri

Programma 2.5

```

/*programma che esegue la divisione tra due float. Utilizza variabili
  locali ed effettua il passaggio di parametri alla funzione chiamata*/
#include "stdio.h"
float dividi(float,float);      /*prototipo della funzione con i parametri*/
main( )    {                  /*inizio del programma principale*/
float num1,num2,risul;        /*variabili locali per la "main( )"*/
printf ("\n INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE  ");
scanf ("%f",&num1);          /*attende che da tastiera venga digitato
                                numero e lo assegna alla variabile num1*/
printf ("INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE  ");
scanf ("%f",&num2);          /*preleva il secondo numero assegnandolo a
                                num2*/
risul = dividi(num1,num2);     /*chiama la funzione, passa i due parametri
                                e assegna il risultato alla variabile
                                locale risul*/
printf ("\n IL RISULTATO DELLA DIVISIONE E'  %f \n",risul);
getch();
    }                          /*fine del programma principale*/
float dividi(float numero1,float numero2)
    {                          /*funzione che esegue la divisione*/
    float risultato;          /*variabile locale a cui viene assegnato il
                                valore del risultato*/
    risultato = numero1 / numero2; /*viene eseguita la divisione e il risultato
                                assegnato alla variabile locale risultato*/
    return risultato;         /*viene restituito al programma principale il
                                risultato della divisione*/
    }

```

Osservazioni

- ▶ Questa volta la funzione *dividi()* non è più dichiarata vuota in quanto ritorna un valore di tipo *float*.
- ▶ Alla funzione vengono passati i due parametri che assumono il valore che hanno al momento della chiamata. Tali valori sono trasferiti alla funzione chiamata che li utilizza.
- ▶ Viene usata l'istruzione *return* per far ritornare al programma chiamante il valore voluto.
- ▶ Ai parametri presenti nella funzione (*numero1* e *numero2*) è stato assegnato un nome diverso da quello presente nella chiamata (*num1* e *num2*); si poteva, volendo, mantenere lo stesso nome.
- ▶ I parametri passati all'atto della chiamata debbono essere rigorosamente corrispondenti a quelli presenti nella funzione (sia per posizione, sia per tipo).

■ Programma con uso di funzioni con passaggio parametri

Programma 2.6

```
#include "stdio.h"
/*programma che esegue la divisione tra due float. Utilizza una variabile globale, che contiene il risultato della divisione, variabili locali ed effettua il passaggio di parametri alla funzione chiamata*/
float risul;                                /*variabile globale */
void dividi(float, float);                  /*prototipo della funzione con i parametri*/
main( ) {                                  /*inizio del programma principale*/
    float num1, num2;                      /*variabili locali per la "main( )"*/
    printf ("INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE ");
    scanf ("%f", &num1);                  /*attende che da tastiera venga digitato il numero e lo assegna alla variabile num1*/
    printf ("INTRODUCI UN NUMERO CON LA VIRGOLA MOBILE ");
    scanf ("%f", &num2);                  /*preleva secondo numero assegnandolo a num2*/
    dividi(num1, num2);                    /*chiama la funzione, passa i parametri*/
    printf ("\n IL RISULTATO DELLA DIVISIONE E' %f ", risul);
    getch();
}                                           /*fine del programma principale*/
void dividi(float numero1, float numero2)
{                                           /*funzione che esegue la divisione*/
    risul = numero1 / numero2;            /*viene eseguita la divisione che modifica il valore della variabile globale risul*/
}                                           /*fine della funzione*/
```

Osservazioni

- La funzione `dividi()` è dichiarata *void* in quanto non restituisce alcun valore, ma modifica la variabile globale *risul* assegnandovi il valore del risultato della divisione.
- La funzione `main()` utilizza il valore della variabile globale, modificato durante la chiamata, per visualizzare il risultato della divisione.

2.8 Le strutture di controllo in C

Nell'esecuzione di un programma normalmente le istruzioni vengono eseguite una dopo l'altra. Esistono però particolari istruzioni che permettono di alterare il flusso con cui il programma viene svolto facendo eseguire un'istruzione piuttosto che un'altra.

2.8.1 Le strutture condizionali (di selezione)

■ Struttura *if*

In questa struttura selettiva viene posta una condizione, se essa è verificata si esegue un'istruzione, o un blocco di istruzioni, altrimenti il programma procede dalla prima istruzione dopo il blocco. Questa struttura in C viene implementata con *if*.

La sintassi del comando è la seguente:

```
if (condizione)
{blocco istruzioni;}
```

Se è presente una sola istruzione le parentesi graffe possono essere omesse e l'istruzione viene posta sulla stessa riga di *if*.

■ Struttura **if ... else**

Una seconda forma, più versatile, della struttura condizionale è rappresentata dalla struttura **if ... else** che utilizza la seguente sintassi:

```
if (condizione)
{blocco istruzioni;}
else (condizione)
{blocco istruzioni;}
```

dove il *blocco istruzioni* può essere in una o in tutte e due le parti una singola istruzione. Il costrutto appena visto permette di operare la scelta tra due possibilità diverse eseguendo un gruppo o l'altro di istruzioni. Nel caso di istruzioni singole, queste vanno poste sulla stessa riga di *if* ed *else*.

■ Struttura **switch ... case**

Quando bisogna operare scelte multiple invece di utilizzare una serie di *if ... else* anche nel C è presente una struttura condizionale di tipo particolare. Essa è la struttura **switch ... case** che ha la seguente sintassi:

```
switch(variabile tipo int o char)
{
    case valore1:
        blocco istruzioni;
    break;
    case valore2:
        blocco istruzioni;
    break;
    .....
    default:
        blocco istruzioni;
}
```

La struttura ora esaminata controlla che il valore della variabile dopo *switch* coincida con uno dei valori posti dopo *case* e, in caso affermativo, esegue il gruppo di istruzioni fino a quando incontra *break*. Se non si verifica alcuna uguaglianza viene saltata l'intera struttura. La parola chiave **default**, può essere messa alla fine del costrutto qualora si vogliano eseguire delle istruzioni quando non si verifichi alcuno dei *case*.

Tuttavia, bisogna osservare che con tale costrutto è possibile effettuare solo un controllo di uguaglianza, inoltre la variabile dopo *switch* deve essere di tipo compatibile con valori di tipo **int**.

2.8.2 Le strutture iterative

■ Il loop *for*

La più semplice delle strutture iterative è quella relativa al **for**. La sintassi della struttura è la seguente:

```
for(inizializza var_ind; verifica var_ind; modifica var_ind;)
{ blocco istruzioni; }
```

dove con *var_ind* si è indicata una variabile che viene utilizzata per il controllo dell'intero ciclo. Le parentesi graffe, al solito, possono non essere utilizzate se si deve eseguire una sola istruzione. Tale struttura permette di eseguire un determinato ciclo per un numero prefissato di volte, ovvero esso viene ripetuto fin quando la variabile indice dal valore iniziale impostato non raggiunge quello finale.

■ Il loop *while*

La sintassi della struttura è la seguente:

```
while ( condizione )
{ blocco istruzioni; }
```

Il blocco delle istruzioni viene eseguito ripetutamente, fin quando la condizione posta risulta vera. *È da tenere ben presente che se tale condizione all'entrata del loop fosse falsa, il blocco non sarebbe eseguito neanche una volta.*

Deve essere cura del programmatore rendere vera la condizione prima di entrare nella struttura. Si ricordi inoltre, che se all'interno della struttura, non esiste alcun controllo che, prima o poi, renda falsa la condizione, non si ha uscita dal loop.

■ Il loop *do...while*

La sintassi è la seguente:

```
do
{ blocco istruzioni; }
while ( condizione )
```

Questa volta il blocco delle istruzioni viene eseguito sempre almeno una volta. Infatti il controllo della condizione avviene alla fine del ciclo e il ciclo viene ripetuto fin tanto che la condizione risulta vera.

Per uscire dalla struttura, al suo interno deve essere presente una qualche istruzione che, prima o poi, renda falsa la condizione.

2.8.3 Le istruzioni *break* e *continue*

Le istruzioni *break* e *continue* sono usate all'interno dei *loop* per ottenere particolari condizioni.

Quando all'interno di un ciclo si incontra l'istruzione *break* viene forzata l'uscita dal ciclo stesso senza tenere conto se sia verificata o meno la condizione posta.

Invece, con *continue* il *loop* viene ripreso dall'inizio senza eseguire le istruzioni che seguono il *continue*.

■ Programma con uso della struttura selettiva **if**

Programma 2.7

```
# include "stdio.h"
main( ) {
    int a,b;
    printf ("introduci due numeri interi separati dalla virgola ");
    scanf ("%d,%d",&a,&b);
    if (a > b) printf ("IL PRIMO NUMERO E' PIU' GRANDE");
    if (a < b) printf ("IL PRIMO NUMERO E' PIU' PICCOLO");
    if (a == b) printf ("I DUE NUMERI SONO UGUALI");
    printf ("\n premere un tasto per uscire");
    getch();
}
```

Lo stesso programma è strutturato utilizzando **if... else**.

■ Programma con uso della struttura selettiva **if... else**

Programma 2.8

```
# include "stdio.h"
main( ) {
    int a,b;
    printf ("introduci due numeri interi separati dalla, virgola ");
    scanf ("%d,%d",&a,&b);
    if (a > b) printf ("\n IL PRIMO NUMERO E' PIU' GRANDE");
    else
    if (a < b) printf ("\n IL PRIMO NUMERO E' PIU' PICCOLO");
    else
    printf ("\n I DUE NUMERI SONO UGUALI");
    getch();
}
```

Notare che il **C** associa l'**else** con il più vicino degli **if**.

■ Programma con uso della struttura selettiva **switch**

Programma 2.9

```
#include "stdio.h"
main( ) {
    char car,a,b,c;
    printf ("INTRODUCI LA LETTERA 'a' O 'b' O 'c' \n");
    scanf ("%c",&car);
    switch (car)
    {
        case 'a':
            printf ("HAI INTRODOTTO LA LETTERA a "); break;
        case 'b':
            printf ("HAI INTRODOTTO LA LETTERA b "); break;
        case 'c':
            printf ("HAI INTRODOTTO LA LETTERA c "); break;
        default:
            printf("LA LETTERA INTRODOTTA NON E' QUELLA RICHIESTA");
    }
    getch();
}
```

/*fine switch*/
/*fine main*/

Il programma chiede di introdurre una particolare lettera dell'alfabeto e controlla che sia stata effettivamente introdotta tale lettera. È chiaro che invece di stampare semplicemente i messaggi possono essere svolte, per ogni case azioni più complesse.

Nel listato precedente viene utilizzata la parola chiave default; essa può essere messa alla fine del costrutto qualora si vogliano eseguire delle istruzioni quando non si verifichi alcuno dei case.

■ Programma con uso della struttura iterativa for

Programma 2.10

```

/* PROGRAMMA DI ESEMPIO SULL'UTILIZZAZIONE DELLA STRUTTURA ITERATIVA
FOR - CALCOLA 10 VALORI DELLA TENSIONE DI USCITA (Vu) PER UN
PARTITORE DI TENSIONE CON CARICO ALL'USCITA. SONO ASSEGNATI I VALORI
DELLA TENSIONE D'INGRESSO (Vi), DEI RESISTORI DEL PARTITORE R1 E R2
E DEL RESISTORE DI CARICO Rc - Rc VIENE DIMINUITO OGNI VOLTA DI UN
DECIMO DEL VALORE INIZIALE - UTILIZZA LA FORMULA:

      Rp
Vu= Vi*  -----      CON   Rp = R2//Rc
      R1+Rp
*/
#include "stdio.h";
float pigreco = 3.141592;
int i;
float Res1, Res2, Rescar, Respar, Rescar_ini, Vu, Vi ;

main() {
    printf ("\n CALCOLO DI 10 VALORI DELLA TENSIONE DI USCITA DI UN ");
    printf ("\n PARTITORE DI TENSIONE CON CARICO - SONO ASSEGNATI I VALORI");
    printf ("\n DELLA TENSIONE D'INGRESSO, DEI RESISTORI R1 E R2 DEL ");
    printf ("\n PARTITORE E DI QUELLO DI CARICO Rc - Rc VIENE DIMINUITO ");
    printf ("\n OGNI VOLTA DI UN DECIMO DEL SUO VALORE INIZIALE");
    printf ("\n");
    printf ("\n SCEGLIERE IL VALORE DELLA TENSIONE D'INGRESSO [V] ");
    scanf ("%f", &Vi);
    printf (" SCEGLIERE IL VALORE DELLA RESISTENZA R1 [Ohm] ");
    scanf ("%f", &Res1);
    printf (" SCEGLIERE IL VALORE DELLA RESISTENZA R2 [Ohm] ");
    scanf ("%f", &Res2);
    printf (" SCEGLIERE IL VALORE DELLA RESISTENZA Rc [Ohm] ");
    scanf ("%f", &Rescar);
    Rescar_ini = Rescar;
    for (i=1 ; i <= 10 ; i++)
    {
        Respar = (Res2*Rescar)/(Res2+Rescar); /*CALCOLA PARALLELO CARICO E R2*/
        Vu= Vi*(Respar/(Res1+Respar)); /*CALCOLA TENSIONE D'USCITA*/
        printf ("\n RESISTENZA DI CARICO = %12.3f Ohm ", Rescar);
        printf ("\n TENSIONE D'USCITA = %12.3f V ", Vu);
        Rescar = Rescar-Rescar_ini/10 ;
    } /*fine del loop for */
    getch();
}

```


■ Programma con uso della struttura iterativa while

Programma 2.11

```

/* PROGRAMMA DI ESEMPIO SULL'UTILIZZAZIONE DELLA STRUTTURA ITERATIVA
   WHILE ... - CALCOLA IL VALORE DELLA TENSIONE DI USCITA (Vu) PER
   UN PARTITORE DI TENSIONE ASSEGNATI I VALORI DELLA TENSIONE DI
   INGRESSO (Vi) E DEI RESISTORI R1 E R2 - IL CALCOLO VIENE ESEGUITO
   PER DIVERSI VALORI DI R2 FINO A QUANDO NON SI PONE R2 = 0 -
   E' UTILIZZATA LA FORMULA:
                                   R2
   Vu = Vi x -----
                                   R1 + R2
*/

#include "stdio.h";

float  Res1,Res2,Vi,Vu;

main() {
    printf ("\n CALCOLA IL VALORE DELLA TENSIONE DI USCITA DI UN PARTITORE ");
    printf ("\n DI TENSIONE - RIPETE IL CALCOLO PER VARI VALORI DI R2 - ");
    printf ("\n");
    printf ("\n INSERIRE IL VALORE DELLA TENSIONE D'INGRESSO [V] ");
    scanf ("%f",&Vi);
    printf ("\n INSERIRE IL VALORE DI R1 [Ohm] ");
    scanf ("%f",&Res1);
    printf ("\n");
    Res2 = 1 ; /* PONE R2 DIVERSO DA ZERO IN MODO CHE SIA VERA LA CONDIZIONE
                QUANDO SI ENTRA NEL WHILE */
    while (Res2 != 0)
    {
        printf ("\n PER TERMINARE INSERIRE R2=0 ");
        printf ("\n SCEGLIERE IL VALORE DELLA RESISTENZA R2 [Ohm] ");
        scanf ("%f",&Res2);
        Vu = Vi*(Res2/(Res1+Res2));
        printf ("\n LA TENSIONE D'USCITA DEL PARTITORE E' %5.4f V \n",Vu);
    }
}

```


■ Programma con uso della struttura iterativa `do ... while`

Programma 2.12

```

/*PROGRAMMA DI ESEMPIO SULL'UTILIZZAZIONE DELLA STRUTTURA ITERATIVA
DO ... WHILE - CALCOLA LE CORRENTI I1 E I2 CHE PERCORRONO I RAMI
DI UN PARTITORE DI CORRENTE REALIZZATO CON DUE RESISTORI R1 E R2 POSTI
IN PARALLELO - E' RICHiesta LA CORRENTE TOTALE It E I VALORI DEI DUE
RESISTORI - IL PROGRAMMA TERMINA QUANDO UNO DEI DUE RESISTORI E'
POSTO UGUALE A ZERO - SONO UTILIZZATE LE FORMULE:

          R2                      R1
I1 = It x -----      I2 = It x -----
          R1 + R2              R1 + R2      */

#include "stdio.h";
float  Res1,Res2,It ,I1,I2;
main()
{
    printf ("\n CALCOLA LE CORRENTI I1 E I2 CHE PERCORRONO I RAMI DI UN ");
    printf ("\n PARTITORE DI CORRENTE - R1 E R2 COSTITUISCONO I DUE RAMI");
    printf ("\n E It LA CORRENTE TOTALE ENTRANTE NEL PARTITORE - RIPETE");
    printf ("\n IL CALCOLO PER VARI VALORI DELLE RESISTENZE FINO A QUANDO");
    printf ("\n UNA DELLE DUE NON ASSUME UN VALORE UGUALE A ZERO -");
    printf ("\n");
    printf ("\n SCEGLIERE IL VALORE DELLA CORRENTE TOTALE [A] ");
    scanf ("%f",&It);
    do
    {
        printf ("\n");
        printf ("\n PER TERMINARE INSERIRE R1=0 O R2=0");
        printf ("\n");
        printf ("\n SCEGLIERE IL VALORE DELLA RESISTENZA R1 [Ohm] ");
        scanf ("%f",&Res1);
        printf ("\n SCEGLIERE IL VALORE DELLA RESISTENZA R2 [Ohm] ");
        scanf ("%f",&Res2);
        if (((Res1==0) && (Res2==0))) /*SE R2=0 E R1=0 NON EFFETTUA CALCOLO*/
        {
            I1 = It*(Res2/(Res1+Res2));
            I2 = It*(Res1/(Res1+Res2));
            printf ("\n LA CORRENTE CHE PERCORRE R1 E' %9.6f A" , I1);
            printf ("\n LA CORRENTE CHE PERCORRE R2 E' %9.6f A" , I2);
        }
    } while (((Res1 != 0) && (Res2 != 0))); /*SE Res1 = 0 O Res2 = 0 ESCE */
}

```

2.9 Array

Permettono di memorizzare più dati (dello stesso tipo) utilizzando un unico *identificatore* ed uno o più *indici* per distinguere i vari valori. Sono **array** a una sola dimensione quelli con un solo indice. Così per esempio `num[0]`, `num[1]`, `num[2]` sono tre variabili appartenenti all'insieme `num[i]` e al variare dell'indice *i* da 0 a 2 esse possono assumere tre diversi valori. L'indice deve essere di tipo *intero* o un'espressione che fornisce un valore intero. Nella dichiarazione iniziale dell'array è necessario indicare quale è il *tipo di dati* che deve essere memorizzato in esso e il numero massimo di elementi (*dim*) che si prevede debbano essere utilizzati nel corso del programma.

La sintassi per la dichiarazione di un array a una sola dimensione è la seguente:

```
tipo_dati nome [dim]
```

Esempi:

```
int num[10]; char lettera[20];
```

La dichiarazione degli *array* a due dimensioni è fatta nel modo seguente:

```
tipo_dati nome [dim1] [dim2]
```

dove con *tipo dati* è indicato il tipo di dati che compongono gli *elementi dell'array*; *nome* è il nome che si assegna all'array e *dim1* e *dim2* sono le sue due dimensioni.

Gli array a due dimensioni possono essere visti come una tabella di dati in cui sono evidenziate le *righe* e le *colonne*. In tali matrici occorrono appunto due *variabili indice* per scandire tutti gli elementi.

Per esempio si disponga dell'*array* dei seguenti numeri interi:

12	34	55	67
3	2	21	55
11	15	8	99

esso dovrà essere dichiarato in **C** nel seguente modo:

```
int numeri [3] [4];
```

La prima dimensione è riferita alle righe e la seconda alle colonne; se le variabili indice sono "i" per le righe e "j" per le colonne, ogni elemento della matrice potrà essere raggiunto con la variabile `numeri[i][j]` assegnando ad *i* tutti i valori che vanno da 0 a 2 e a *j* quelli che vanno da 0 a 3.

Così l'elemento 12 (primo elemento), sarà localizzato dalla variabile `numeri[0][0]` e l'elemento 99 (ultimo elemento) da `numeri[3][4]`.

Poiché il C non effettua alcun controllo sui dati immessi bisogna essere ben sicuri che il tipo di dato dichiarato per il vettore sia quello giusto e che il numero di variabili con indice utilizzate non ecceda quello assegnato.

2.9.1 Stringhe

Un particolare uso degli *array* a una dimensione è quello legato con le *stringhe* ovvero un insieme di caratteri alfanumerici. Nel C non esiste un dato primario di tipo *stringa*; queste sono rappresentate con un *array monodimensionale di caratteri* (tipo *char*). Così per definire una stringa formata da non più di quindici caratteri si deve effettuare la seguente dichiarazione:

```
char nome[16];
```

come si vede è stato necessario riservare 16 posti di tipo *char* e non 15, perché in C una stringa deve sempre terminare col carattere *nul*, cioè 00h (in C tale carattere può essere anche indicato con `\0`).

Per introdurre stringhe da tastiera, nel C sono utilizzate due funzioni della libreria del linguaggio: `scanf()` e `gets()`. Con queste funzioni il carattere finale viene automaticamente posizionato al termine della stringa; tuttavia nella dichiarazione esso deve essere preso in considerazione come se fosse un elemento della stringa.

Sono presenti poi in C un certo numero di funzioni atte a manipolare le stringhe. Le più utilizzate sono (si noti nella colonna a destra della tabella l'header che le rende eseguibili):

tabella 2.14

Funzioni che operano sulle stringhe			
tipo	funzione	descrizione	header
<i>uns. int</i>	strlen (<i>string1</i>)	fornisce la lunghezza di <i>string1</i> (è escluso il carattere terminatore)	string.h
<i>char</i>	strcpy (<i>string1</i> , <i>string2</i>)	copia <i>string2</i> in <i>string1</i>	string.h
<i>int</i>	strcmp (<i>string1</i> , <i>string2</i>)	confronta <i>string1</i> e <i>string2</i> (restituisce zero se le stringhe sono uguali)	string.h
<i>char</i>	strcat (<i>string1</i> , <i>string2</i>)	collega <i>string1</i> con <i>string2</i> .	string.h
<i>double</i>	atof (<i>const char *s</i>)	converte una stringa in numero in virgola mobile	stdlib.h
<i>int</i>	atoi (<i>const char *s</i>)	converte una stringa in numero <i>int</i>	stdlib.h
<i>int</i>	toascii (<i>int n</i>)	converte <i>n</i> (<i>int</i>) nel range 0-127	ctype.h
	itoa (<i>int n</i> , <i>char *s</i> , <i>int b</i>)	converte <i>n</i> nella stringa <i>s</i> con formato base <i>b</i> .	string.h

Osservazioni

- Nella funzione `strcpy()` nella dichiarazione di *string1* questa deve essere lunga almeno come *string2*.
- Nella funzione `strcat()` nella dichiarazione di *string1* quest'ultima deve essere lunga abbastanza da contenere anche *string2*.

■ Programma con uso di un array monodimensionale

Programma 2.13

```
#include "stdio.h"
main() {
    int i, numero, num[6];
    for (i = 1; i <= 5; i++)
    {
        printf ("\n INTRODUCI  %d° NUMERO  ",i);
        scanf ("%d",&numero);
        num[i]=numero;
    }
    /*fine ciclo for*/

    for(i=1; i<=5;i++)
    { printf("\n %d° numero %d ", i,num[i] );}
    getch();

}
/*fine main()*/
```

Osservazioni

- Il modo in cui è stato usato `printf()` permette di scrivere ogni volta, prima dell'immissione del numero, "INTRODUCI IL 1° NUMERO", poi "INTRODUCI IL 2° NUMERO" e così via; infatti, di volta in volta, al carattere di formattazione "%d" viene sostituito il valore della variabile "i". Le variabili adoperate sono:

`num[1], num[2], num[3], num[4], num[5]`

- Con `scanf()` si è acquisita la variabile `numero` e, solo dopo, il suo valore è stato assegnato alla variabile con indice. Si poteva acquisire direttamente con `scanf()` la variabile con indice.

■ Programma con acquisizione di stringhe da tastiera

Programma 2.14

```
#include "stdio.h"
main() {
    char string1[21], string2[11];
    printf("INTRODUCI UNA STRINGA DI LUNG. MAX. 20 CARATTERI  ");
    gets(string1);
    printf("INTRODUCI UNA STRINGA DI LUNG. MAX. 10 CARATTERI  ");
    gets(string2);
    printf("\n LE STRINGHE INTRODOTTE SONO :\n %s \n %s",string1,string2);
    getch();

}
/*fine main()*/
```

Nell'introduzione delle stringhe da tastiera è stata utilizzata la funzione `gets()` in sostituzione di `scanf("%s", ...)`. In realtà **l'utilizzazione di `scanf()` nell'acquisizione di stringhe, potrebbe in alcuni casi, produrre effetti non desiderati.**

Si può ora precisare una particolarità della funzione `scanf()` utilizzata nell'introduzione delle stringhe; per acquisire una stringa da tastiera si deve scrivere:

```
char stringa[21];
.....
scanf ("%s",stringa);
```


Come si vede, contrariamente a quanto succede con gli altri tipi di dato, la parola *stringa* non è preceduta dal simbolo "&" in quanto essa, senza parentesi né indice, rappresenta già un indirizzo. Nella dichiarazione iniziale si presuppone che la stringa possa contenere non più di 20 caratteri; si sono riservati 21 posti per allocare correttamente anche il terminatore della stringa (carattere NUL) che verrà automaticamente aggiunto dalla funzione `scanf()`.

2.10 I puntatori

Il linguaggio C fa un grande uso dei *puntatori*. Si definisce puntatore un indirizzo della memoria atto a referenziare un dato in essa contenuto. In questo modo le variabili vengono individuate per mezzo dell'indirizzo che esse hanno in memoria e non attraverso l'*identificatore* a esse associato. Naturalmente anche l'indirizzo verrà indicato con un nome (*identificatore*) e non con un numero.

Si può pertanto affermare che **un puntatore è una variabile alla quale è associato l'indirizzo di un'altra variabile**. Per dichiarare una variabile come puntatore è usata la seguente sintassi:

tipo_dato *identificatore

dove con `tipo_dato` si è indicato uno dei dati primari del C (cioè: `char`, `int`, `float`, `double` o `void`), e con *identificatore* il nome che si assegna al "puntatore". Il simbolo "*" è quello che indica che si sta dichiarando una variabile di tipo puntatore.

Il programma 2.15 illustra l'uso dei puntatori.

■ Programma con uso dei puntatori

Programma 2.15

```
#include "stdio.h"
/*CHIEDE DI INTRODURRE DUE NUMERI INTERI, DI CIASCUNO DI ESSI STAMPA IL
 VALORE E L'INDIRIZZO - SI PUO' VEDERE CHE GLI INDIRIZZI DIFFERISCONO
 DI DUE LOCAZIONI IN QUANTO UN INTERO OCCUPA IN MEMORIA DUE BYTE */
main() {
    int num1,num2,*punt_num;
    punt_num = &num1;      /*assenga al puntat. l'indiriz. di memoria di num1*/
    printf ("\n INTRODUCI UN NUMERO INTERO ");
    scanf ("%d",&num1);    /*nell'indirizzo assegnato viene posto il numero*/
    printf ("\n IL NUMERO INTRODOTTO E':          %d  ",*punt_num);
    printf ("\n L'INDIRIZZO DEL NUMERO INTRODOTTO E': %u\n",&num1);
    punt_num = &num2;      /*assenga al puntat. l'indiriz. di memoria di num2*/
    printf ("\n INTRODUCI UN ALTRO NUMERO INTERO ");
    scanf ("%d",&num2);    /*nell'indirizzo assegnato viene posto il numero*/
    printf ("\n IL NUMERO INTRODOTTO E':          %d  ",*punt_num);
    printf ("\n L'INDIRIZZO DEL NUMERO INTRODOTTO E': %u  ",&num2);
    getch();
}
```

Nota

Nel programma sono stati adoperati i due operatori relativi ai puntatori "*" e "&". L'**operatore &, che restituisce l'indirizzo di memoria della variabile a cui è applicato**, è stato già precedentemente utilizzato con la funzione `scanf()`. L'**operatore *, invece, imposta il valore della variabile che si trova all'indirizzo referenziato dal puntatore**.

La dichiarazione iniziale `*punt_num`, serve per riservare memoria alla variabile puntatore, ma non inizializza il puntatore stesso. Perché tale puntatore possa

venir utilizzato è necessario effettuare l'inizializzazione; tale operazione è svolta con l'istruzione `punt_num = &num1` ovvero viene assegnato al puntatore `punt_num` l'indirizzo della variabile intera `num1`. In tal caso con `punt_num` non deve essere usato l'operatore "*", altrimenti verrebbe indicato il valore della variabile puntata. Quindi si ha:

- ▶ `int num1` dichiara una variabile di tipo intero;
- ▶ `int *punt_num` dichiara un puntatore a un dato di tipo intero;
- ▶ `punt_num = &num1` inizializza il puntatore assegnando a esso l'indirizzo della variabile `num1`;
- ▶ `*punt_num` indica il valore della variabile contenuta all'indirizzo referenziato dal puntatore.

2.11 Funzioni aritmetiche

Nella tabella 2.15 sono descritte alcune funzioni aritmetiche e viene anche indicato il relativo file d'intestazione.

tabella 2.15

tipo	funzione	descrizione	Header
int	abs (int <i>n</i>)	Ritorna il valore assoluto dell'intero <i>n</i>	stdlib.h
double	acos (double <i>n</i>)	Ritorna l'arcocoseno in radianti di <i>n</i> (-1 + 1)	math.h
double	asin (double <i>n</i>)	Ritorna l'arcoseno in radianti di <i>n</i> (-1 + 1)	math.h
double	atan (double <i>n</i>)	Ritorna l'arcotangente in radianti di <i>n</i>	math.h
double	atan2 (double <i>y</i> , double <i>x</i>)	Ritorna l'arcotangente in radianti di <i>y/x</i>	math.h
double	ceil (double <i>n</i>)	Ritorna l'intero più piccolo non minore di <i>n</i> (arrotondamento per eccesso)	math.h
double	cos (double <i>alfa</i>)	Ritorna il coseno dell'angolo <i>alfa</i> espresso in radianti	math.h
double	exp (double <i>n</i>)	Ritorna il valore di <i>e</i> (costante di Nepero) elevato a <i>n</i>	math.h
double	fabs (double <i>n</i>)	Ritorna il valore assoluto di <i>n</i>	math.h
double	floor (double <i>n</i>)	Ritorna l'intero più grande non maggiore di <i>n</i> (arrotondamento per difetto)	math.h
double	hypot (double <i>m</i> , double <i>n</i>)	Ritorna l'ipotenusa di un triangolo rettangolo con lati <i>m</i> e <i>n</i>	math.h
double	log (double <i>n</i>)	Ritorna il logaritmo in base <i>e</i> di <i>n</i>	math.h
double	log10 (double <i>n</i>)	Ritorna il logaritmo in base 10 di <i>n</i>	math.h
double	modf (double <i>n</i> , double <i>*i</i>)	Ritorna la parte frazionata di <i>n</i> e memorizza nella locazione puntata da <i>i</i> la parte intera di <i>n</i>	math.h
double	pow (double <i>x</i> , double <i>y</i>)	Ritorna il valore di <i>x</i> elevato alla <i>y</i> (x^y)	math.h
double	pow10 (int <i>x</i>)	Ritorna il valore di 10 elevato alla <i>x</i> (10^x)	math.h
int	random (int <i>n</i>)	Ritorna un numero casuale compreso tra 0 e <i>n</i> -1	math.h
double	sin (double <i>alfa</i>)	Ritorna il seno dell'angolo <i>alfa</i> espresso in radianti	math.h
double	sqrt (double <i>n</i>)	Ritorna la radice quadrata di <i>n</i> (<i>n</i> deve essere ≥ 0)	math.h
double	tan (double <i>alfa</i>)	Ritorna la tangente dell'angolo <i>alfa</i> in radianti	math.h

2.12 Metodi di compilazione dei programmi in C

I programmi sorgente presenti in questa Unità possono essere provati con il **C++ Builder 6.0** in ambiente Windows XP o anche con il nuovo **C++ Builder XE** (della Embarcadero), in ambiente Windows Vista o Windows 7.

Il file sorgente è rappresentato dal programma scritto con un editore di testo, in formato ASCII, senza formattazione. Il sorgente, per divenire eseguibile, deve essere opportunamente compilato, servendosi del compilatore proprio del linguaggio che si sta adoperando.

In rete sono poi reperibili versioni di compilatori rilasciati sotto licenza GPL (*General Public License*) con i quali sviluppare i programmi come ad esempio **Dev-C++** (<http://bloodshed.net>). Questo compilatore pur non essendo più aggiornato è ugualmente adatto alla compilazione dei programmi presenti in questa unità. Per Dev-C++ è possibile trovare in rete anche dei tutorial in italiano.

Il **Pelles C compiler** è un altro compilatore *free* che è possibile scaricare da Internet e che permette di compilare ed eseguire anche i file in C standard (e non solo in C++). È dotato di un ambiente IDE facilmente utilizzabile.

Si tenga presente che alcuni caratteri speciali inseriti nei listati potrebbero cambiare in base al set di caratteri in uso con il sistema operativo con cui si sta operando.

■ Utilizzazione di Dev C++.

L'ultima versione stabile del compilatore è la 4.9.9.2 (disponibile anche in italiano). Una volta scaricata eseguire un'installazione tipica. È possibile installare l'applicazione anche in Windows 7. Avviata l'applicazione si apre l'IDE (*Integrated Development Environment*) di Dev-C++. Per eseguire la compilazione dei file sorgente degli esempi si utilizzano i vari menu presenti sulla barra superiore:

- menu FILE ⇒ NEW ⇒ SOURCE FILE;
- menu FILE ⇒ OPEN PROJECT OR FILE;
- ricercare il file da compilare, selezionarlo ed aprirlo;
- avviare la compilazione e l'esecuzione con il menu EXECUTE ⇒ COMPILE & RUN

Con Dev C++ si deve aggiungere insieme a `#include "stdio.h"` anche `#include "conio.h"`.

■ Pelles C compiler

Il prodotto si installa con molta facilità avviando il file *setup.exe* (è presente anche una versione a 64 bit *setup64.exe*). Seguendo quindi le istruzioni a video:

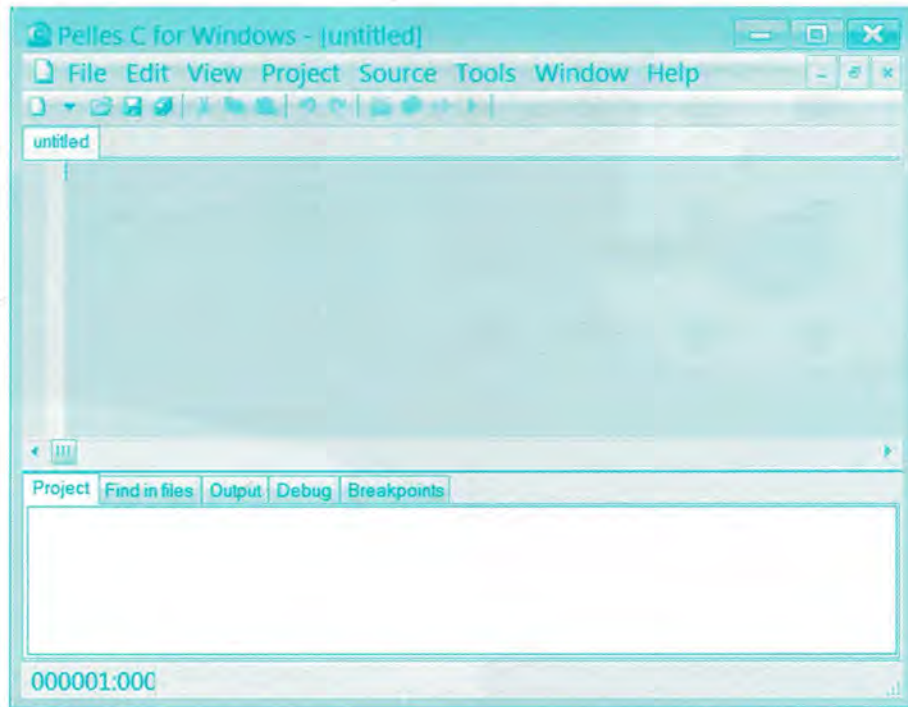
- si scelga un'installazione di tipo **NORMAL**;
- terminata l'installazione conviene creare un collegamento sul desktop;
- si lanci l'applicazione con doppio clic sulla relativa icona;
- nella finestra BROWSE INFORMATION che si apre, digitare il pulsante CLOSE.

L'IDE (*Integrated Development Environment*) del compilatore è mostrato nella **figura 2.1**.

L'applicazione, durante l'installazione, crea in DOCUMENTI la cartella PELLE C PROJECTS in cui (nella sotto cartella SAMPLES ⇒ STANDARD C) possono essere posti i sorgenti in C da compilare. Se i file sorgente sono posti in un'altra cartella è necessario scegliere il percorso per caricarli.

Prima di effettuare la compilazione del primo file è utile eseguire alcune impostazioni sulla finestra a cartelle che si apre con il menu TOOLS ⇒ OPTIONS:

- nella cartella GENERAL togliere il segno di spunta su SAVE FILE WITHOUT PROMPTING BEFORE BUILD e su LOAD LAST PROJECT AT STARTUP (ed, eventualmente, su MAKE BACKUP COPIES OF SOURCES FILES).
- Nella cartella SOURCE, selezionare poi il tipo di carattere desiderato e la grandezza necessaria per avere una buona visualizzazione.



Per la compilazione degli esempi proposti nell'unità debbono essere apportate ai file le seguenti modifiche:


- eliminare (se presente) o mettere a commento la funzione `getch()`;
- inserire alla fine del programma una riga con la funzione `printf("\n");` (chiusa dal punto e virgola), se si desidera che il messaggio *press any key to continue ...*, visualizzato automaticamente, non sia attaccato al testo scritto dall'applicazione in esecuzione.

Le operazioni da compiere per compilare un programma sono le seguenti:

- con il menu **FILE** ⇒ **OPEN** scegliere il percorso in cui sono stati posti i file sorgente;
- selezionare il programma e premere il pulsante **OPEN**; Il file viene caricato e mostrato sull'IDE;
- con il menu **PROJECT** ⇒ **COMPILE FILE** o **BUILD FILE** (o con il corrispondenti pulsanti posti sulla barra degli strumenti) compilare il file sorgente; se ci sono errori, prima di procedere, debbono essere corretti;
- nella finestra **NEW DEFAULT PROJECT** scegliere **WIN32 CONSOLE PROGRAM (EXE)** e dare **Ok**;
- con il menu **PROJECT** ⇒ **EXECUTE FILE** (o con il corrispondente pulsante posto sulla barra degli strumenti) avviare l'esecuzione del programma.

Si noti che il compilatore per ogni file `.c` caricato durante la compilazione, crea una serie di file di progetto compreso un file eseguibile (`.exe`).

- Con **C++ Builder 6.0** si opera nel seguente modo:

1. dal menu **FILE** ⇒ **NEW** ⇒ **OTHER ...** (o con il pulsante **NEW** ) aprire la finestra **NEW ITEMS**, selezionare **CONSOLE WIZARD** e dare **Ok**. Impostare le opzioni come in **figura 2.1** e **figura 2.2** e poi ancora **Ok**.

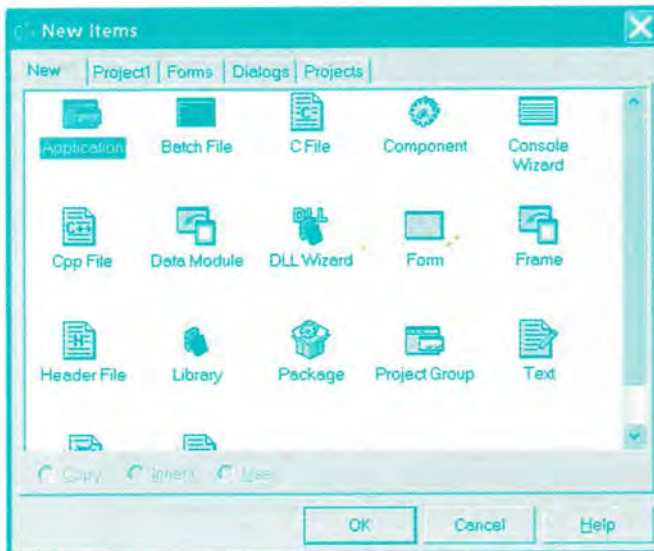


figura 2.1

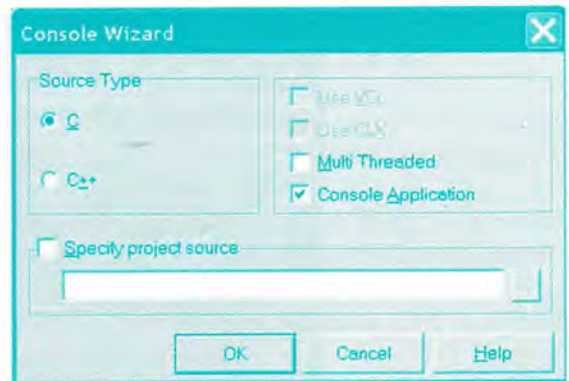


figura 2.2

2. Viene visualizzata la finestra UNIT1.C. Selezionare e quindi cancellare tutto il suo contenuto.
3. Aprire il file in C che si vuole eseguire con il menu FILE ⇒ OPEN:
 - con CERCA IN selezionare la cartella in cui si trova il file;
 - in TIPO FILE scegliere *C file* (*.cpp, *.hpp, *.c, *.h);
 - nell'elenco di file che viene visualizzato scegliere quello da aprire;
 - il file aperto viene posto in una nuova cartella con il nome del file.
4. Copiare interamente nella cartella UNIT1.C il file aperto:
 - Selezionare tutto il file;
 - EDIT ⇒ COPY;
 - EDIT ⇒ PASTE.
5. Avviare l'esecuzione del programma con il tasto funzione F9 o con il pulsante di RUN.

Alla fine del file, prima della parentesi graffa, si deve inserire (se non presente) la funzione `getch()`; (incluso il punto e virgola).

La funzione `getch()` blocca l'esecuzione del programma in attesa che l'utente prema un tasto qualsiasi. Senza di essa, la console su cui vengono visualizzati i dati in uscita sarebbe visualizzata per un tempo brevissimo.

Chiudendo C++ Builder non eseguire il salvataggio del progetto (a meno che non si voglia salvare il programma in prova come progetto di C++ Builder).

► Con **C++ Builder Xe** si procede con le stesse modalità di C++ Builder 6.