IL LINGUAGGIO C++

I linguaggi di programmazione

I linguaggi di programmazione permettono di scrivere algoritmi interpretabili da un sistema di elaborazione. Un algoritmo scritto in un linguaggio di programmazione viene chiamato programma e il processo di scrittura del programma, a partire dall'algoritmo, viene chiamato codifica. I linguaggi di programmazione sono costituiti da un alfabeto e da un insieme di regole che devono essere rispettate per scrivere programmi sintatticamente corretti.

Il linguaggio macchina costituito da zero ed uno è l'unico che pilota direttamente le unità fisiche dell'elaboratore in quanto è l'unico comprensibile dall'elaboratore stesso. È però estremamente complicato scrivere programmi in tale linguaggio naturale per la macchina ma completamente innaturale per l'uomo. Per poter permettere un dialogo più semplice con la macchina sono nati i linguaggi di programmazione.

Il linguaggio C e il C++

Nel 1972, presso i Bell Laboratories, Dennis Ritchie progettava e realizzava la prima versione del linguaggio C.

Il C si distingueva dai suoi predecessori per il fatto di implementare una vasta gamma di tipi di dati (carattere, interi, numeri in virgola mobile, strutture) non originariamente previsti dagli altri due linguaggi. Da allora ad oggi il C ha subito trasformazioni: la sua sintassi è stata affinata, soprattutto in conseguenza della estensione object-oriented (C++). Il C++, come messo in evidenza dallo stesso nome, rappresenta una evoluzione del linguaggio C: il suo progettatore (Bjarne Stroustrup) quando si pose il problema di trovare uno strumento che implementasse le classi e la programmazione ad oggetti, invece di costruire un nuovo linguaggio di programmazione, pensò bene di estendere un linguaggio già esistente, il C appunto, aggiungendo nuove funzionalità

Elementi lessicali

Ogni programma scritto in un qualsiasi linguaggio di programmazione prima di essere eseguito viene sottoposto ad un processo di compilazione o interpretazione (a seconda che si usi un compilatore o un interprete). Lo scopo di questo processo è quello di tradurre il programma originale (codice sorgente) in uno semanticamente equivalente, ma eseguibile su una certa macchina. Il processo di compilazione è suddiviso in più fasi, ciascuna delle quali volta all'acquisizione di opportune informazioni necessarie alla fase successiva.

La prima di queste fasi è nota come analisi lessicale ed ha il compito di riconoscere gli elementi costitutivi del linguaggio sorgente, individuandone anche la categoria lessicale. Ogni linguaggio prevede un certo numero di categorie lessicali e in C++ possiamo distinguere in particolare le seguenti categorie lessicali:

- Commenti;
- Identificatori;
- Parole riservate;
- Costanti letterali;
- Segni di punteggiatura e operatori;

a) Commenti

E' importantissimo dire che ogni programma ben realizzato dovrebbe includere dei commenti. Nel realizzare un commento si deve rispettare l'intelligenza dei programmatori e nel contempo non dare tutto troppo per scontato. Si tenga presente che spesso un programmatore è costretto a "mettere le mani" nel codice scritto da altri sviluppatori e, se il codice non è ben commentato, tale operazione può rivelarsi molto più ardua del previsto. Infine, i commenti sono utili anche per l'autore del programma stesso: pensate a cosa accadrebbe se doveste riprendere il codice scritto da voi stessi tre anni fa!

Dunque in C++ una linea di codice commentata è preceduta dalla doppia barra //. Esiste, anche un secondo tipo di commento: il **commento a blocchi**, ereditato dal C. In questo caso, un commento inizia con i simboli /* e termina con i simboli */. La differenza sostanziale è che il commento a blocchi permette di commentare più linee di codice

senza dover ripetere ad ogni linea i simboli del commento stesso.

Ma vediamo un esempio per comprendere meglio.

```
// PRIMO.CPP
// Il primo esempio in C++

/*
  PRIMO.CPP
  Il primo esempio in C++
*/
```

Semplice, no? I due tipi di commenti sono assolutamente intercambiabili. E' ovvio che se avete la necessità di commentare parecchie linee di codice consecutive è conveniente ricorrere al commento a blocchi.

b) Identificatori

Gli identificatori sono simboli definiti dal programmatore per riferirsi a cinque diverse categorie di oggetti:

- Variabili;
- Costanti simboliche;
- Etichette;
- Tipi definiti dal programmatore;
- Funzioni;

Un identificatore è formato da una sequenza di una o più lettere, cifre o caratteri e deve iniziare con una lettera o con un carattere di sottolineatura. Gli identificatori possono contenere un qualunque numero di caratteri ma solo i primi 31 caratteri sono significativi per il compilatore.

Il linguaggio C++ è case sensitive, cio' vuol dire che il compilatore considera le lettere maiuscole e minuscole come caratteri distinti. Ad esempio le variabili MAX e max saranno considerati come due identificatori distinti e, quindi, rappresenteranno due differenti celle di memoria.

Ecco alcuni esempi di identificatori:

```
i
MAX
max
first_name
_second_name
```

E' fortemente consigliato utilizzare degli identificatori che abbiano un nome utile al loro scopo. Ovvero, se un identificatore dovrà contenere l'indirizzo di una persona sarà certamente meglio utilizzare il nome indirizzo piuttosto che il nome casuale XHJOOQQO. Sono entrati a far parte della programmazione comune gli identificatori i, j, k che vengono spesso utilizzati come contatori nelle iterazioni.

1) Variabili

Una **variabile** non è nient'altro che un contenitore di informazione che viene utilizzata, poi, da qualche parte in un programma C++. Naturalmente, per poter conservare una determinata informazione, tale contenitore deve far uso della memoria del computer.

Come lo stesso nome suggerisce facilmente, una variabile, dopo essere stata dichiarata, può modificare il suo contenuto durante l'**esecuzione** di un programma. Il responsabile di tali eventuali modifiche altri non è che il programmatore il quale, tramite opportune istruzioni di assegnamento impone che il contenuto di una variabile

possa contenere una determinata informazione. Possiamo tranquillamente dire che le variabili rappresentano l'essenza di qualunque programma di computer. Senza di esse, i computer diventerebbero totalmente inutili. Ma cosa intendiamo quando parliamo di "**dichiarazione di una variabile**"? Diciamo che è necessario fornire al computer una informazione precisa sul tipo di variabile che vogliamo definire; per esempio, se stiamo pensando di voler assegnare ad una variabile un valore numerico dovremo fare in modo che il computer sappia che dovrà allocare una certa quantità di memoria che sia sufficiente a contenere tale informazione in modo corretto e senza possibilità di equivoci. A tale scopo, il C++ fornisce una serie di "tipi" standard che permettono al programmatore di definire le variabili in modo opportuno a seconda della tipologia dell'informazione che si vuole conservare. Vediamo dunque quali sono i tipi standard del C++.

I tipi standard del C++

Per la stragrande maggioranza dei casi, i **sette tipi base del C++** sono sufficienti per rappresentare le informazioni che possono essere manipolate in un programma. Vediamo, allora, quali sono tali tipi: testo o **char**, intero o **int**, valori in virgola mobile o **float**, valori in virgola mobile doppi o **double**, enumerazioni o **enum**, non-valori o **void** e puntatori.

I tipi di dati primari, eccetto il tipo void, ammettono dei modificatori di tipo. Tali modificatori: signed, unsigned, long, short (con segno, senza segno,lungo e corto) sono impiegati per adattare con maggiore precisione i tipi di dati fondamentali alle esigenze del programmatore.

Tipo	Dim. in bit	Valori ammessi	Utilizzo
char	8	da -128 a 127	Numeri piccoli e caratteri ASCII
unsigned char	8	da 0 a 255	Piccoli numeri e il set di caratteri del PC
bool	8	0 (falso) o 1 (vero)	Indicatori del verificarsi di eventi
int	16	da -32.768 a +32.767	Contatori, piccoli numeri, controllo-cicli
int	32	da -2.147.483.648 a 2.147.483.647	Contatori, piccoli numeri, controllo-cicli
long int	32	da -2.147.483.648 a 2.147.483.647	Grandi numeri
float	32	da $3{,}4x10^{-38}$ a $3{,}4x10^{38}$	Notazione Scientifica (7 cifre di precisione)
double	64	da $1{,}7x10^{-308}$ a $1{,}7x10^{308}$	Notazione Scientifica (15 cifre di precisione)

È bene fare qualche osservazione sulla tabella:

- I tipi signed (char, int, long) conservano in memoria dati utilizzando il metodo del complemento a 2
- Il tipo float e il tipo double sono rappresentazioni floating-point: vengono comunemente indicati, rispettivamente, come numeri a singola precisione e a doppia precisione
- Si noti che il tipo char viene utilizzato sia per conservare piccoli numeri che per conservare il set di caratteri, cioè dati apparentemente completamente diversi fra di loro dal punto di vista della elaborazione. La cosa è meno strana di quello che può sembrare a prima vista, basta riflettere sulla circostanza che, anche se si tratta di caratteri, la conservazione in memoria avviene associando ad ogni carattere una configurazione binaria e, quindi, un numero. Si può allora considerare la stessa configurazione come numero o come carattere corrispondente a quella configurazione binaria: la configurazione 0100.0001

(secondo per esempio la codifica ASCII) può essere il numero 65 come anche il carattere 'A'. Tutto ciò può tornare utile per certi tipi di elaborazione sui caratteri.

- La specificazione int, se accompagnata da un modificatore, può essere omessa. Si può quindi dichiarare una variabile indifferentemente long int o solo long

Per poter conoscere la dimensione in byte, che il compilatore che si sta usando, utilizza per la conservazione in memoria di un dato di un certo tipo, si può usare la funzione sizeof():

```
#include <iostream>
using namespace std;
int main(){
  cout << "\nDimensione di un char   " << sizeof(char);
  cout << "\nDimensione di un bool   " << sizeof(bool);
  cout << "\nDimensione di un int    " << sizeof(int);
  cout << "\nDimensione di un float  " << sizeof(float);
  cout << "\nDimensione di un double " << sizeof(double);
  return 0;
}
```

Il programma mostra a video la dimensione in byte di un dato del tipo specificato come parametro della sizeof.

2) Costanti simboliche

Le costanti simboliche servono ad identificare valori che non cambiano nel tempo, non possono essere considerate dei contenitori, ma solo un nome per un valore.

Costanti definite (#define)

Possiamo usare la direttiva **#define** del preprocessore per dare un nome ad una costante nel seguente modo:

#define *identificatore_di_costante*

Ovunque useremo l'identificatore, il preprocessore provvederà a sostituirlo con la costante. Ad esempio

#define PI 3.14159265

#define NEWLINE '\n'

Questo ci evita di dover riscrivere la stessa costante più volte in posti diversi del programma con la possibilità di commettere errori quali scrivere in qualche posto **3.14159365** invece di **3.14159265**. Una volta che abbiamo associato un identificatore ad una costante possiamo usare tale identificatore in ogni punto seguente del programma come se esso fosse una costante. Ad esempio:

circonferenza = 2 * PI * r;

cout << NEWLINE;

La direttiva **#define** non è una istruzione C++ ma una direttiva per il preprocessore. Essa deve quindi essere scritta in una sua propria riga e non deve essere aggiunto il carattere punto e virgola (**;**) alla fine.

3) Etichette

Una etichetta è un nome il cui compito è quello di identificare una istruzione del programma e sono utilizzate dall'istruzione di salto incondizionato goto.

4) Tipi definiti dal programmatore

Un tipo invece, come vedremo meglio in seguito, identifica un insieme di valori e di operazioni definite su questi valori; ogni linguaggio fornisce un certo numero di tipi primitivi (cui è associato un identificatore di tipo predefinito) e dei meccanismi per permettere la costruzione di nuovi tipi (a cui il programmatore deve poter associare un nome) a partire da quelli primitivi.

5) Funzioni

Infine funzione è il termine che il C++ utilizza per indicare i sottoprogrammi.

c) Parole riservate

Ogni linguaggio si riserva delle parole chiave (keywords) il cui significato è prestabilito e che non possono essere utilizzate dal programmatore come identificatori. Il C++ non fa eccezione:

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	trow	while

Sono inoltre considerate parole chiave tutte quelle che iniziano con un doppio underscore __; esse sono riservate per le implementazioni del linguaggio e per le librerie standard e il loro uso da parte del programmatore dovrebbe essere evitato in quanto non sono portabili.

d) Costanti letterali.

Una costante è una qualsiasi espressione che ha un valore prefissato. Esse si possono suddividere in Numeri Interi, Numeri in Virgola Mobile, Caratteri e Stringhe.

Numeri Interi

1776
+707
-273

sono letterali che denotano numeri interi decimali. Per scrivere una costante intera non occorre usare le virgolette (") o qualche altro carattere speciale. Non ci sono dubbi sul fatto che i letterali precedenti denotino delle costanti: quando scriviamo **1776** in un programma intendiamo proprio il valore 1776 e non altri.

Oltre ai numeri in notazione decimale (quella comunemente usata) il C++ accetta anche letterali che denotano numeri interi ottali (base 8) e numeri interi esadecimali (base 16). Per scrivere un numero in notazione ottale basta premettere il carattere **0** (carattere zero) e per scrivere un numero in notazione esadecimale occorre premettere i due caratteri **0x** (zero e x). Ad esempio i seguenti letterali sono equivalenti:

75 ***// decimale***

0113 ***// ottale***

0x4b ***// esadecimale***

Essi denotano lo stesso numero: 75 (settantacinque) espresso rispettivamente come numero in base 10, in base 8 e in base 16.

Nota: le cifre decimali sono le usuali (0,1,2,3,4,5,6,7,8,9), le cifre ottali sono soltanto (0,1,2,3,4,5,6,7) mentre come cifre esadecimali usiamo (0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f).

Numeri in virgola mobile

Sono numeri con una parte frazionaria e/o un fattore esponenziale. Sono rappresentati con letterali in cui compare il punto decimale che separa la parte intera dalla parte decimale (noi usiamo la virgola ma nei paesi anglosassoni

si usa il punto) e/o un carattere **e** seguito da un esponente intero (che si legge "per 10 alla X, dove X è l'intero che segue il carattere **e**").

3.14159 *// 3.14159*

6.02e23 *// 6.02 x 10^{23}*

1.6e-19 *// 1.6 x 10^{-19}*

3.0 *// 3.0*

sono tutti letterali che rappresentano numeri in virgola mobile.

Caratteri e stringhe

Vi sono anche dei letterali che rappresentano costanti non numeriche:

'z'
'p'
"Salve gente"

"Come state?"

I primi due rappresentano un singolo carattere mentre gli altri due rappresentano stringhe di diversi caratteri. Osserviamo che i letterali che rappresentano singoli caratteri sono racchiusi tra due caratteri apice (') mentre i letterali che rappresentano stringhe sono racchiusi tra due caratteri doppio apice ("). Questo è necessario per poter distinguere valori di tipo stringa da valori di tipo carattere (che sono considerati diversi). Inoltre, l'uso degli apici (') e (") evita di confondere un letterale carattere o stringa da un identificatore o una parola chiave. Infatti in:

x
'x'

x denota la variabile **x**, mentre **'x'** denota la costante di tipo carattere **'x'** .

Per rappresentare con un letterale (o all'interno di un letterale stringa) alcuni caratteri speciale si usano notazioni particolari (**i codici di escape**). Ecco una lista di tali codici di escape (ognuno di essi inizia con il carattere barra rovesciata (\):

\n	a capo riga
\r	ritorno carrello
\t	tabulazione
\v	tabulazione verticale
\a	allerta (beep)
\'	apice singolo (')
\"	doppio apice (")
\?	punto interrogativo (?)
\\	barra rovesciata (\ )

Possiamo inoltre rappresentare ogni carattere del codice ASCII usando il suo codice numerico in ottale preceduto da una sbarra rovesciata (\) oppure il suo codice numerico in esadecimale preceduto da una sbarra rovesciata e un carattere x (**\x**). Ad esempio **\23** e **\37** rappresentano i caratteri ASCII di codice 19 e 31 mentre, usando la notazione esadecimale, i medesimi due caratteri si denotano con **\x13** e **\x1f** .

e) Segni di punteggiatura e operatori

Alcuni simboli sono utilizzati dal C++ per separare i vari elementi sintattici o lessicali di un programma o come operatori per costruire e manipolare espressioni.

Operazioni e espressioni

1) Dichiarazione delle variabili

Prima di usare una variabile occorre dichiararla specificando a quale tipo di dato essa appartenga. La sintassi di una dichiarazione di variabile prevede prima il nome del tipo di dato (quale **int**, **short**, **float** ...) seguito dall'identificatore scelto per denotare tale variabile. Ad esempio:

```
int a;
float numero;
```

Sono dichiarazioni di variabili corrette. La prima dichiara una variabile di tipo **int** denotata dall'identificatore **a**. La seconda dichiara una variabile di tipo **float** denotata dall'identificatore **numero**. Una volta dichiarate, le variabili **a** e **numero** possono essere usate nel programma all'interno del loro campo di validità (lo scope).

Se vogliamo dichiarare più di una variabile dello stesso tipo possiamo farlo in una stessa riga indicando una sola volta il tipo e separando gli identificatori con la virgola. Ad esempio:

int a, b, c;

dichiara tre variabili (**a**, **b** e **c**) di tipo **int**, ed ha esattamente lo stesso significato di:

```
int a;
int b;
int c;
```

a) Costanti dichiarate (const)

Usando il prefisso **const** si possono dichiarare delle costanti appartenenti ad un determinato tipo esattamente allo stesso modo in cui si dichiarano le variabili:

```
const int larghezza = 100;
const char tab = '\t';
const int cap =  12440;
```

In realtà le costanti dichiarate sono semplicemente delle variabili il cui valore non può più essere modificato (il compilatore controlla che in nessun punto del programma compaia una assegnazione o una qualsiasi altra istruzione che può modificare il valore di una costante e in tal caso segnala un errore). Naturalmente, siccome una volta create non è più possibile cambiarne il valore, esse devono essere sempre inizializzate con un valore al momento della loro creazione.

Confronto fra la direttiva #define e la specifica di tipo const

Mentre con l'uso di const:

- il tipo della costante é dichiarato; un eventuale errore di dichiarazione viene segnalato immediatamente
- la costante é riconosciuta, e quindi analizzabile, nelle operazioni di debug
- la costante può essere definita localmente ad una funzione

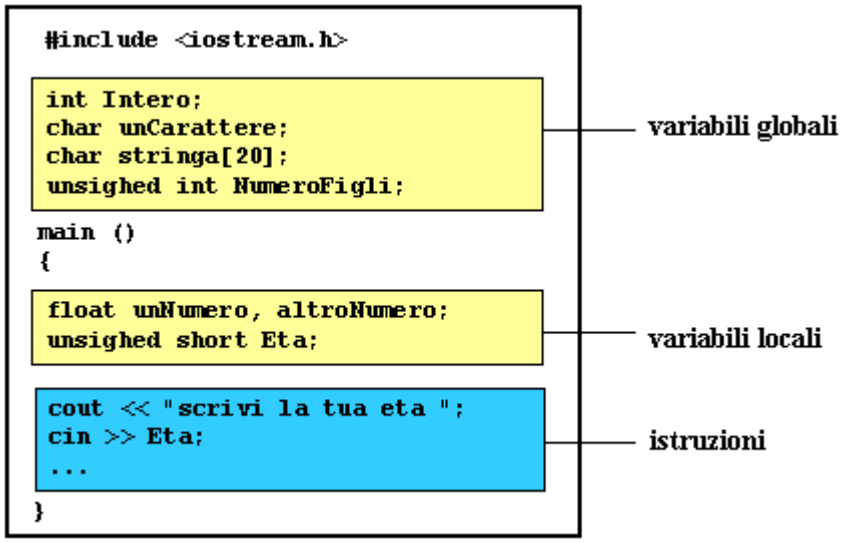
con #define:

- viene definita una costante a livello globale, senza allocarla a in memoria con il suo nome.

b) Scope delle variabili

Tutte le variabili che vogliamo usare devono essere preventivamente dichiarate. Una differenza importante tra C e C++ è che in C++ possiamo dichiarare delle variabili in ogni punto del programma, anche tra due istruzioni eseguibili, e non solo all'inizio di un blocco di istruzioni come è richiesto in C.

Resta comunque normalmente preferibile seguire le indicazioni del C per dichiarare le variabili in quanto è comodo, durante la fase di correzione del programma (debugging), avere le dichiarazioni raggruppate assieme: all'inizio di ogni funzione (per le variabili locali) o direttamente nel corpo del programma al di fuori di ogni funzione (variabili globali).



Le variabili globali si possono usare in tutto il programma dal punto in cui sono dichiarate fino alla fine.

Lo scopo delle **variabili locali** è invece limitato al blocco in cui sono dichiarate. Se sono dichiarate all'inizio di una funzione (come in main) il loro scopo è l'intero corpo della funzione. Quedto significa che se nell'esempio ci fosse un'altra funzione diversa da **main()**, le variabili locali dichiarate in **main** non sarebbero visibili all'interno di tale funzione e viceversa.

In C++ lo scopo di una variabile locale è limitato alla parte del blocco in cui esse sono dichiarate che segue la dichiarazione stessa (un blocco è un gruppo di istruzioni racchiuse tra parentesi graffe **{}**.

Oltre allo scopo **locale** e **globale** esiste anche uno scopo **esterno** (**external**) che non solo è visibile in tutto il file che contiene il programma ma è visibile anche in ogni altro file di programma che venga collegato con esso.

2) Inizializzazione delle variabili

Quando dichiariamo una variabile il suo valore è indeterminato (i bit della zona di memoria riservata alla variabile hanno il valore che era stato loro assegnato da qualche precedente programma). Potremmo volere che una variabile abbia un valore particolare fin dal momento in cui viene dichiarata. Per fare questo basta aggiungere alla dichiarazione un simbolo di uguale seguito dal valore desiderato:

***tipo identificatore* = *valore_iniziale* ;**

Ad esempio se vogliamo dichiarare una variabile **a** di tipo **int** con valore iniziale **0** possiamo scrivere:

int a = 0;

Oltre a questo modo di inizializzare le variabili (noto come stile C), vi è un altro modo più consono allo stile C++: racchiudendo il valore iniziale tra parentesi tonde

***tipo identificatore*(*valore_iniziale*);**

Ad esempio:

int a(0); Il C++ accetta entrambe le notazioni.

3) Operazioni sulle variabili

Una volta appreso dell'esistenza delle variabili e delle costanti possiamo iniziare ad operare con esse. A questo scopo il C++ fornisce degli operatori, che nel nostro linguaggio sono un insieme di parole chiave e simboli speciali. E' importante conoscerli perché essi sono la base del linguaggio C++.

Non è necessario inparare a memoria tutto il contenuto di questa sezione. La maggior parte dei dettagli sono riportati per poterli consultare in seguito qualora se ne abbia bisogno.

Assegnamento (=).

L'operatore di assegnamento serve per assegnare un valore ad una variabile.

```
lvalue = rvalue;
```

a = 5;

assegna il valore intero **5** alla variabile **a** . La parte alla sinistra dell'operatore = è nota come *lvalue* (left value) e la parte destra *rvalue* (right value). *lvalue* deve essere sempre una variabile mentre la parte destra può essere una costante, una variabile, il risultato di una operazione o una qualsiasi combinazione di essi.

Occorre notare che l'operatore di assegnamento opera sempre da destra verso sinistra e non nel senso inverso.

a = b;

assegna alla variabile **a** (*lvalue*) il valore della variabile **b** (*rvalue*) indipendentemente dal valore che era precedentemente memorizzato nella variabile **a** . Possiamo pensare l'*lvalue* di **a** come l'indirizzo della zona di memoria riservata per memorizzare il valore di **a** mentre l'*rvalue* di **b** lo dobbiamo pensare come il valore memorizzato nella zona di memoria riservata per memorizzare il valore di **b**. Notiamo inoltre che noi stiamo soltanto assegnando ad **a** il valore di **b** e che una modifica successiva di **b** non cambia il nuovo valore di **a**.

Ad esempio, il seguente codice (in cui è evidenziata in verde come commento l'evoluzione del contenuto delle variabili):

```
int a, b;   // a:?  b:?
a = 10;     // a:10 b:?
b = 4;      // a:10 b:4
a = b;      // a:4 b:4
b = 7;      // a:4 b:7
```

otteniamo alla fine che il valore contenuto in **a** è **4** e quello contenuto in **b** è **7**. L'ultima modifica di **b** non ha modificato **a**, benché appena prima avessimo scritto **a = b**.

Una caratteristica dell'operatore di assegnamento presente in C++ ma non in altri linguaggi di programmazione è che un assegnamento si può usare come *rvalue* (o parte di un *rvalue*) in un altro assegnamento. Ad esempio:

a = 2 + (b = 5);

è equivalente a:

```
b = 5;
a = 2 + b;
```

ossia: prima assegna **5** alla variabile **b** e poi assegna ad **a** il valore **2** più il risultato del precedente assegnamento a **b** (**5** appunto), ottenendo **7** come valore finale di **a**. Quindi, anche la seguente espressione è valida in C++:

a = b = c = 5;

assegna **5** a tutte e tre le variabili **a**, **b** e **c**.

Operatori aritmetici (+, -, *, /, %)

I cinque operatori aritmetici previsti dal linguaggio sono:

- **+** somma
- **-** differenza
- ***** moltiplicazione
- **/** divisione
- **%** modulo (o resto)

Gli operatori di somma, differenza, moltiplicazione e divisione denotano le usuali quattro operazioni numeriche. L'operatore di modulo fornisce il resto della divisione di due numeri interi. Ad esempio, se scriviamo **a = 11 % 3;**, viene assegnato alla variabile **a** il valore **2** in quanto **2** è proprio il resto che si ottiene dividendo 11 per 3.

Operatori di assegnamento composti (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Gli operatori di assegnamento composti sono una caratteristica del C++ che contribuisce alla sua fama di essere un linguaggio sintetico. Essi permettono di modificare il valore di una variabile con una sola operazione:

valore += incremento;	è equivalente a **valore = valore + incremento;**
a -= 5;	è equivalente ad **a = a - 5;**
a /= b;	è equivalente ad **a = a / b;**
prezzo *= numero + 1;	è equivalente a **prezzo = prezzo * (numero + 1);**

e analogamente per le altre operazioni.

Incremento e decremento.

Un altro esempio di sinteticità si ha con gli operatori di incremento (++) e di decremento (--). Essi aumentano o diminuiscono di 1 il valore di una variabile e sono equivalenti a **+= 1** e **-= 1** rispettivamente. Quindi:

```
a++;
a += 1;
a = a+1;
```

fanno la stessa cosa: aumentano di 1 il valore di **a**.
Questi operatori si possono usare sia come *prefissi* che come *postfissi* . Ossia possono essere scritti prima della variabile (**++a**) o dopo di essa (**a++**). Benché in espressioni semplici come **a++** o **++a** gli operatori prefissi e postfissi abbiano lo stesso significato, in altri casi in cui viene utilizzato il risultato dell'operazione nella valutazione di un'altra espressione essi assumono un significato diverso: Se l'operatore di incremento viene usato come *prefisso* (**++a**) il valore della variabile viene incrementato prima della valutazione dell'espressione e quindi l'espressione viene valutata usando il valore incrementato; se l'operatore di incremento viene usato come *postfisso* (**a++**) il valore della variabile viene incrementato dopo la valutazione dell'espressione e quindi l'espressione viene valutata usando il valore non incrementato. Ecco un esempio della differenza tra i due modi di usare l'operatore:

Esempio 1

```
B = 3;
A = ++B;
```

Equivale a:

```
B=B+1
A=B
// A è 4, B è 4
```

Esempio 2

```
B = 3;
A = B++;
```

Equivale a:

```
A=B
B=B+1
// A è 3, B è 4
```

Nel primo esempio, la variabile **B** viene incrementata prima che il suo valore venga copiato in **A** mentre nel secondo esempio viene prima copiato in **A** il valore della variabile **B** e poi viene incrementato il valore della variabile **B**.

Operatori relazionali (==, !=, >, <, >=, <=)

Per confrontare i valori di due espressioni si usano gli operatori relazionali. Come specificato dallo standard ANSI-C++, il risultato di un operatore relazionale è di tipo **bool** e può assumere soltanto uno dei due valori booleani **true** o **false**, a seconda del risultato del confronto.

Ecco la lista degli operatori relazionali del C++:

==	Uguale
!=	Diverso
>	Maggiore
<	Minore
>=	Maggiore o uguale
<=	Minore o uguale

Ed ecco alcuni esempi:

(7 == 5) risultato **false** .
(5 > 4) risultato **true** .
(3 != 2) risultato **true** .
(6 >= 6) risultato **true** .
(5 < 5) risultato **false** .

naturalmente, invece di usare soltanto costanti numeriche possiamo usare qualunque espressioni valida, comprese le variabili. Supponiamo che **a=2**, **b=3**e **c=6**,

(a == 5) risultato **false**.
(a*b >= c) risultato **true** in quanto (2*3 >= 6).
(b+4 > a*c) risultato **false** in quanto (3+4 > 2*6).
((b=2) == a) risultato **true**.

Attenzione. L'operatore = (un solo uguale) è differente dal simbolo == (doppio uguale), il primo è l'operatore di assegnamento (assegna il valore dell'espressione alla sua destra alla variabile alla sua sinistra e ritorna tale valore) mentre il secondo è l'operatore relazionale di uguaglianza che confronta i valori delle due espressioni che stanno ai suoi lati e ritorna il valore booleano **true** o **false** a seconda che esse abbiano lo stesso valore o valori diversi. Quindi nell'ultima espressione **((b=2) == a)**, viene dapprima assegnato il valore **2** a **b** e quindi tale valore viene confrontato con il valore di **a**, che è pure **2**, e quindi il risultato che si ottiene è il valore **true**.

Operatori logici (!, &&, ||).

L'operatore **!** è l'operatore logico di negazione NOT. Esso ha un unico operando (di tipo **bool**) posto alla sua destra e il suo risultato è l'opposto del valore dell'operando: se l'operando è **true** esso ritorna **false** , se l'operando è **false** esso ritorna **true** . Ad esempio:

!(5 == 5) ritorna **false** in quanto l'espressione alla sua destra (5 == 5) è **true** .

!(6 <= 4) ritorna **true** in quanto (6 <= 4) è **false** .

!true ritorna **false**.

!false ritorna **true**.

Gli operatori **&&** e || sono gli operatori logici di congiunzione (AND) e disgiunzione (OR). Essi hanno due operandi (di tipo **bool**), uno alla sinistra ed uno alla destra, e il risultato è quello riportato nella seguente tabella:

Primo operando **a**	Secondo operando **b**	Risultato di **a && b**	Risultato di **a || b**
true	true	**true**	**true**
true	false	**false**	**true**
false	true	**false**	**true**
false	false	**false**	**false**

Ad esempio:

((5 == 5) && (3 > 6)) ritorna **false** (*true* && *false*). **((5 == 5) || (3 > 6))** ritorna **true** (*true* || *false*) .

Operatore condizionale (?).

L'operatore condizionale valuta una espressione booleana e ritorna un valore diverso a seconda che tale valore sia **true** oppure **false**. La sua forma è:

condizione ? risultato1 : risultato2

se condizione è **true** l'espressione ritorna *risultato1*, altrimenti ritorna *risultato2*.

7==5 ? 4 : 3 ritorna **3** perché **7** non è uguale a **5** .

7==5+2 ? 4 : 3 ritorna **4** perché **7** è uguale a **5+2** .

5>3 ? a : b ritorna **a**, perché **5** è maggiore di **3** .

a>b ? a : b ritorna il maggiore dei due, **a** o **b** .

Operatori bit a bit (&, |, ^, ~, <<, >>).

Gli operatori bit a bit operano in parallelo su tutti i bit degli operandi. Essi sono operatori di basso livello a cui corrispondono alcune operazioni assembler. Nella programmazione normale C++ essi non dovrebbero essere usati.

operatore	assembler	Descrizione
&	**AND**	AND bit a bit
|	**OR**	OR bit a bit

^	**XOR**	OR esclusivo bit a bit
~	**NOT**	NOT bit a bit (complemento a uno)
<<	**SHL**	Spostamento dei bit (shift) a sinistra
>>	**SHR**	Spostamento dei bit (shift) a destra

Operatori espliciti di conversione di tipo (casting)

Gli operatori di conversione servono per convertire valori appartenenti ad un tipo in valori di un altro tipo. Vi sono diversi modi per fare questo in C++. Il vecchi modo (stile C) consiste nel far precedere le espressioni che devono essere convertite dal nome del nuovo tipo racchiuso tra parentesi **()**:

```
int i;
float f = 3.14;
i = (int) f;
```

Il codice precedente converte il numero float **3.14** nel valore intero **3**. L'operatore di conversione è rappresentato da **(int)**. Un modo più consono allo stile C++ è quello di usare la funzione *costruttore*: far precedere l'espressione da convertire, racchiusa tra parentesi, dal nome del nuovo tipo.

```
i = int ( f );
```

Operatore sizeof()

Questo operatore ha un parametro che può essere sia il nome di un tipo che una espressione. Esso ritorna la memoria, in byte, necessaria a memorizzare un valore di tale tipo o il valore dell'espressione:

```
a = sizeof (char);
```

ritorna **1** in **a** perché un valore di tipo **char** occupa un byte.
Il valore ritornato da è una costante. Esso è quindi determinato a tempo di compilazione (prima dell'esecuzione del programma).

Priorità degli operatori

Quando scriviamo espressioni complesse con molti operatori e operandi possono sorgere dei dubbi sull'ordine in cui gli operatori vengono valutati. Ad esempio, con l'espressione:

```
a = 5 + 7 % 2
```

può sorgere il dubbio tra le seguenti due interpretazioni:

a = 5 + (7 % 2) con risultato **6**, o
a = (5 + 7) % 2 con risultato **0**

L'interpretazione corretta è la prima, quella con risultato **6**. Vi è un ordine di priorità prestabilito tra tutti gli operatori (sia aritmetici che non aritmetici). L'ordine di priorità è il seguente:

L'*associatività* specifica in quale ordine vengono valutati gli operatori con la stessa priorità: partendo da quello più a destra o da quello più a sinistra.

La precedenza degli operatori in una espressione si può modificare o rendere evidente usando le parentesi **(** e **)** come nell'esempio seguente:

```
a = 5 + 7 % 2;
```

si può scrivere come:

a = 5 + (7 % 2);oppure
a = (5 + 7) % 2;

a seconde di come si vuole venga eseguita l'espressione. Pertanto, quando si vuole scrivere una espressione complicata e non si è certi della precedenza degli operatori è bene mettere le parentesi; questo renderà oltretutto più leggibile l'espressione.

Struttura di un programma C++

Probabilmente il modo migliore per iniziare lo studio di un linguaggio di programmazione è cominciare con un programma. Ecco quindi il nostro primo programma:

```
//il mio primo programma in C++
#include <iostream>
Using namespace std;
int  main()
{
cout <<"salve gente!";
system("PAUSE");
return 0;
}
```

Salve gente!

Nella parte sinistra è riportato il codice del nostro primo programma che noi chiameremo, ad esempio, salve.cpp. La parte destra mostra quello che si ottiene eseguendolo. Il modo con cui si scrive, compila ed esegue un programma dipende dal sistema operativo, dall'editore di testi e dal compilatore che abbiamo a disposizione. (in questo caso si è usato il compilatore Dev-C++)

Il precedente programma è il primo programma che la maggior parte dei principianti scrive ed il risultato che si ottiene è l'apparire della scritta **Salve gente** sullo schermo. E' uno dei più semplici programmi che si possano scrivere in C++ ma esso già contiene le principali componenti di un qualsiasi programma C++. Vediamole una alla volta:

// il mio primo programma in C++

Questa è una riga di commento. Tutte le righe che iniziano con due sbarrette (//) vengono considerate commento e non hanno alcun effetto sul comportamento del programma. Esse possono essere usate dal programmatore per includere nel codice del programma alcune brevi spiegazioni ed osservazioni. Nel nostro caso la riga contiene una breve spiegazione di ciò che il programma deve fare.

#include <iostream.h>

Le frasi che iniziano con il simbolo di cancelletto (#) sono direttive per il preprocessore del compilatore. Esse non sono istruzioni eseguibili ma soltanto indicazioni per il compilatore. Nel nostro caso la frase **#include <iostream.h>** dice al preprocessore del compilatore di includere il file della libreria standard **iostream.h** . Questo particolare file contiene le dichiarazioni delle operazioni basilari di input-output definite nella libreria standard del C++ e viene incluso perché tali operazioni serviranno in seguito nel programma. Nei compilatori più recenti, si utilizza il comando **using namespace std;** che permette di indicare solo il nome della libreria senza estensione**..#include <iostream>.**

int main ()

Questa riga è l'inizio della dichiarazione della funzione **main**. La funzione **main** è il punto da cui inizia l'esecuzione di un qualsiasi programma C++. E' irrilevante il punto del programma in cui compare tale funzione - essa è sempre la prima ad essere eseguita. Ovviamente è indispensabile che ogni programma contenga una funzione **main**.

main è seguito da una coppia di parentesi **()** perché esso è una funzione. In C++ tutte le funzioni sono seguite da una coppia di parentesi **()** che, eventualmente, possono contenere degli argomenti. Subito dopo la dichiarazione (l'intestazione) della funzione viene il contenuto (il corpo) della funzione racchiuso tra parentesi graffe (**{}**).

cout << "Salve gente!";

Questa istruzione effettua la cosa più importante del programma **cout** è il flusso standard di output del C++ (di solito indirizzato allo schermo), e l'effetto dell'istruzione è appunto quello di inserire una sequenza di caratteri (nel nostro caso "Salve gente!") in tale flusso di output. La dichiarazione di **cout** si trova nel file iostream.h, ed è per questo che abbiamo dovuto includere tale file. Notiamo che la frase finisce con un punto e virgola (;). Il punto e virgola indica la fine dell'istruzione e deve essere messo alla fine di ogni istruzione (uno degli errori più comuni è appunto dimenticare il punto e virgola alla fine di una istruzione.

System ("PAUSE");

serve a bloccare l'esecuzione del programma; chiede di premere un tasto per continuare.

return 0;

L'istruzione **return** fa terminare la funzione **main()** e ritorna quello che è indicato di seguito, nel nostro caso **0**. Questo è il modo normale di terminare un programma la cui esecuzione è avvenuta senza errori. Come vedremo nei prossimi esempi, tutti i programmi C++ finiscono con una frase simile a questa. (Molti compilatori inseriscono automaticamente una istruzione **return** alla fine di una funzione.)

Dunque, non tutte le righe di un programma denotano una azione. Vi sono righe che contengono soltanto commenti (quelle che iniziano con //), righe che contengono istruzioni per il precompilatore (quelle che iniziano con #), righe che iniziano la dichiarazione di una funzione (nel nostro caso, la funzione **main**) e infine righe che contengono istruzioni eseguibili (come **cout << "Salve gente";**), queste ultime sono contenute nel corpo della funzione (il blocco delimitato dalle parentesi graffe (**{}**) della funzione **main** .

Il programma è stato diviso in più righe per renderlo più leggibile, ma questo non è necessario. Ad esempio, invece di

```
int main ()
{
  cout << "Salve gente";
  system("PAUSE");
  return 0;
}
```

avremmo potuto scrivere:

```
int main () { cout << "Salve gente"; system("PAUSE"); return 0; }
```

tutto nella stessa riga e questo avrebbe avuto esattamente lo stesso significato.

In C++ la separazione tra istruzioni è indicata dal punto e virgola (;) che termina ciascuna di esse. La suddivisione del programma in più righe serve a rendere il programma più leggibile alle persone che lo leggono, per il compilatore la cosa non fa' alcuna differenza.

Comunicazione da console.

La *console* è l'interfaccia base del calcolatore, essa è di solito composta da una tastiera ed un video. La tastiera viene usata come unità di *input* standard e il video come unità di output *output* standard.

Nalla libreria standard *iostream* del C++ le operazioni di *input* ed *output* di un programma vengono gestite da due flussi di dati (*stream*): **cin** per l'input e **cout** per l'output. Sono definiti inoltre altri due flussi - **cerr** e **clog** - il cui scopo è quello di segnalare eventuali messaggi di errore. Tali flussi possono essere mandati anch'essi sul video oppure inviati in un file di *log* .

Dunque **cout** (il flusso di output standard) è normalmente diretto al video e **cin** (il flusso di input standard) è normalmente assegnato alla tastiera.

Usando questi due flussi un programma può interagire con un utente mostrando messaggi sullo schermo e ricevendo l'input da parte dell'utente dalla tastiera.

Output (cout)

Il flusso **cout** viene usato assieme all'operatore sovraccaricato << (una coppia di segni di *"minore"*).

```
cout << "Frase di output"; // stampa Frase di output sullo schermo
cout << 120;              // stampa il numero 120 sullo schermo
cout << x;                // stampa il valore della variabile x sullo schermo
```

L'operatore << è noto come *operatore di inserimento* in quanto esso inserisce il valore alla sua destra nel flusso di dati indicato alla sua sinistra. Negli esempi precedenti esso inserisce nel flusso di output standard **cout** la stringa costante *Frase di output*, la costante numerica **120** e il valore della variabile **x**. Osserviamo che nella prima riga la frase da stampare è racchiusa tra doppi apici ("), questo perché essa è una stringa di caratteri. Quando vogliamo usare una stringa costante di caratteri dobbiamo racchiuderla tra doppi apici (") per distinguerla da un identificatore di variabile. Ad esempio, le due frasi seguenti hanno un significato molto diverso:

```
cout << "Salve";     // stampa Salve sullo schermo
cout << Salve;       // stampa il valore della variabile Salve sullo schermo
```

L'operatore di *inserzione* (<<) può essere usato più di una volta nella stessa frase. Ad esempio:

```
cout << "Salve, " << "io sono " << "una frase C++";
```

stampa sullo schermo **Salve, io sono una frase C++** . L'utilità di poter usare più volte l'operatore di inserzione nella stessa frase si vede quando dobbiamo stampare una combinazione di costanti e variabili o anche semplicemente più di una variabile:

```
cout << "Salve, io ho " << eta << " anni e il mio CAP e' " << cap;
```

Se assumiamo che la variabile **eta** abbia il valore **24** e che la variabile **cap** abbia valore **65064** l'output che si ottiene è:

```
Salve, io ho 24 anni e il mio CAP e' 65064
```

Attenzione: **cout** non va a capo dopo aver stampato a meno che non glielo si dica esplicitamente. Pertanto, con le due seguenti frasi:

```
cout << "Questa e' una frase.";
cout << "Questa e' un'altra frase.";
```

otteniamo:

```
Questa e' una frase.Questa e' un'altra frase.
```

anche se sono state scritte con due distinte chiamate a **cout** . Per andare a capo bisogna inserire esplicitamente nel flusso un carattere speciale di *nuove-linea* che, in C++, si indica con **\n** :

```
cout << "Prima frase.\n ";
cout << "Seconda frase.\nTerza frase.";
```

produce:

```
Prima frase.
Seconda frase.
Terza frase.
```

Per andare a capo possiamo anche usare il *manipolatore* **endl** . Ad esempio:

```
cout << "Prima frase." << endl;
cout << "Seconda frase." << endl;
```

stampa:

```
Prima frase.
Seconda frase.
```

Input (cin).

In C++ l'input standard si effettua applicando l'operatore di *estrazione* (>>) al flusso **cin** . Tale operatore deve essere seguito dalla variabile in cui deve essere memorizzato il valore da leggere. Ad esempio:

```
int eta;
cin >> eta;
```

dichiara la variabile eta di tipo int e quindi aspetta un input da cin (tastiera) per poter memorizzare un valore intero in tale variabile.

Attenzione: **cin** elabora l'input ricevuto da tastiera soltanto dopo che è stato premuto il tasto di invio ENTER. Quindi, anche se viene richiesto di leggere un singolo carattere, **cin** non elabora l'input fino a quando, dopo aver premuto il tasto del carattere da leggere, non viene premuto anche il tasto ENTER.

Quando si usa l'estrattore (>>) su **cin** bisogna tener presente il *tipo* della variabile che si usa per memorizzare il valore letto. Se viene richiesto un intero deve essere introdotto un intero, se viene richiesto un carattere deve essere introdotto un carattere e se viene richiesta una stringa deve essere introdotta una stringa.

```
// esempio di i/o
#include <iostream>
using namespace std;
int main ()
{
  int i;
  cout << "Dammi un intero: ";
  cin >> i;
  cout << "Il valore che mi hai dato e' " << i;
  cout << " e il suo doppio e' " << i*2 << ".\n";
system("PAUSE");
  return 0;
}
```

```
Dammi un intero: 702
Il valore che mi hai dato e' 702 e il suo doppio e' 1404.
```

Un utente malaccorto può essere una delle cause che provocano un errore nell'esecuzione di semplici programmi che usano **cin** (come quello appena visto). La ragione è che se viene richiesto un intero e l'utente introduce il suo nome (che è una stringa di caratterei) il programma non funziona correttamente perché una stringa non è ciò che esso si aspetta dall'utente. Per ora, nei nostri programmi di esempio, assumeremo che l'utente sia perfettamente cooperativo ed introduca sempre dati del tipo richiesto dal programma. Vedremo in seguito come sia possibile scrivere programmi più *robusti* che accettano qualsisi input e, qualora esso non sia del tipo appropriato, chiedono all'utente di riimmettere un valore corretto.

Si può anche usare **cin** per richiedere più di un dato alla volta:

```
cin >> a >> b;
```

è equivalente a:

```
cin >> a;
cin >> b;
```

In entrambi i casi l'utente deve fornire due valori dei tipi appropriati, uno per la variabile **a** ed uno per la variabile **b** . Tali valori possono essere separati da uno o più caratteri *spazio* o *tabulazione* o *nuova-linea* .

Strutture di controllo.

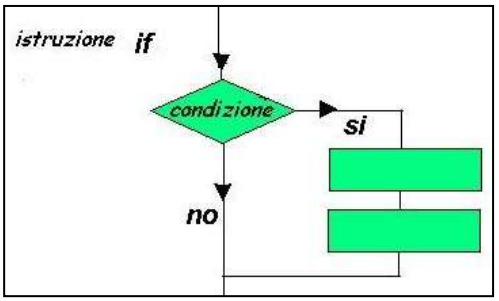
Normalmente un programma non è semplicemente una sequenza di istruzioni da eseguire una dopo l'altra. Durante la sua esecuzione esso può scegliere tra gruppi di istruzioni o ripetere più volte uno stesso gruppo di istruzioni. A questo scopo il C++ fornisce delle strutture di controllo che servono a specificare come si deve comportare il programma.

Per parlare di strutture di controllo occorre introdurre il concetto di ***blocco di istruzioni***. Un blocco di istruzioni è un gruppo di istruzioni separate da punti e virgola (**;**) e racchiuse tra parentesi graffe: **{** e **}** .

Nella maggior parte delle strutture di controllo che vedremo ora viene indicata una (o più) generica ***Istruzione*** come parametro, questa può essere sia una singola istruzione che un blocco di istruzioni. Se mettiamo una singola istruzione non è necessario racchiuderla tra parentesi graffe (**{}**), ma naturalmente occorre mettere il punto e virgola (**;**) che la termina. Se vogliamo mettere più di una istruzione allora occorre racchiuderle tra parentesi graffe (**{}**) per formare un blocco.

a) Strutture condizionali: *if* ed *else*

Viene usata per eseguire una istruzione o un blocco di istruzioni soltanto se si verifica una determinata condizione. La sua forma è:



if (*Condizione*) *Istruzione*

dove ***Condizione*** è una espressione di tipo **bool** (ossia una espressione il cui risultato è uno dei due valori di verità **true** o **false**). L'espressione booleana **Condizione** viene valutata e se essa ritorna il valore **true** l'istruzione (o gruppo di istruzioni) **Istruzione** viene eseguita mentre se essa ritorna il valore **false** l'istruzione **Istruzione** viene ignorata (non eseguita) e l'esecuzione del programma continua con le istruzioni che seguono immediatamente la struttura condizionale.

Ad esempio il seguente frammento di codice stampa **x e' 100** soltanto se il valore della variabile **x** è proprio 100:

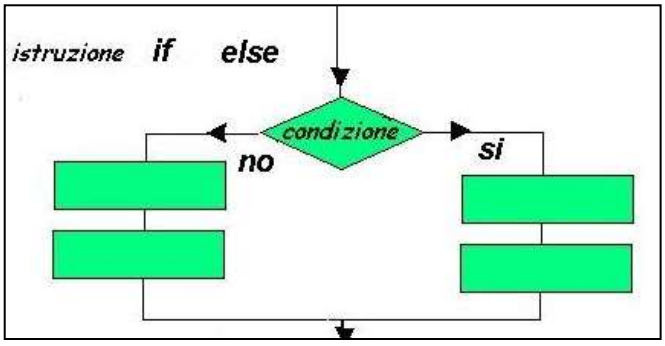
```
if (x == 100)
     cout << "x e' 100";
```

Se vogliamo che vengano eseguite più istruzioni nel caso in cui **Condizione** è **true** dobbiamo racchiuderle tra parentesi graffe per formare un unico *blocco di istruzioni* :

```
if (x == 100)
 {  cout << "x e' ";
  cout << x;  }
```

Possiamo anche usare la parola chiave **else** per specificare una diversa istruzione o blocco di istruzioni da eseguire nel caso in cui la condizione sia falsa. La forma della struttura condizionale è in questo caso la seguente:

if (*Condizione*) *Istruzione1* else *Istruzione2*



Ad esempio:

```
if (x == 100)
   cout << "x e' 100";
else
       cout << "x non e' 100";
```

stampa **x e' 100** se **x** vale proprio 100, ma se non è così -e solo se non è così- stampa **x non e' 100**.

La struttura *if + else* si può concatenare per verificare un insieme di valori. Il seguente esempio illustra il suo uso per verificare se il valore della variabile **x** è negativo, positivo o nessuno dei due (ossia è zero).

```
if (x > 0)
   cout << "x e' positivo";
else if (x < 0)
       cout << "x e' negativo";
else
   cout << "x e' 0";
```

L' operatore condizionale ? (ternary condition) e' la forma piu' efficiente per esprimere semplici if, ha la seguente forma:

<variabile> = <condizione> ? <risultato1> : <risultato2>;

che equivale a:

if (<condizione>) **<variabile>** =<istruzione1>
else **<variabile>** =<istruzione2> ;

Ad esempio:

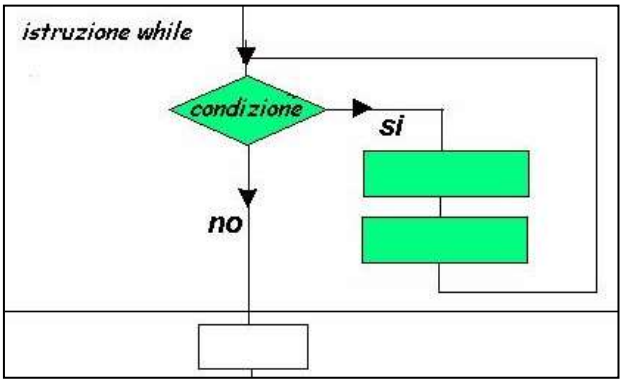
z=(a>b) ? a : b

equivale a:

if (a>b)
z=a;
else
z=b; ovvero assegna a z il massimo tra a e b.

b) Istruzioni iterative o cicli

I *cicli* hanno lo scopo di ripetere una istruzione o gruppo di istruzioni un certo numero di volte oppure fino a che rimane vera una certa condizione.



1) Il ciclo *while*.
Il suo formato è:

while (*Espressione*) *Istruzione*

ed il suo effetto è semplicemente ripetere ***Istruzione*** fintanto che ***Espressione*** ha il valore **true** .

Vediamo, ad esempio, un programma che effettua un conto alla rovescia utilizzando un ciclo *while*:

```
// conto alla rovescia usando while
#include <iostream>

using namespace std;
int main ()
{
  int n;
  cout << "Dammi il valore da cui partire > ";
  cin >> n;
  while (n>0)

{
    cout << n << ", ";
    --n;
  }
  cout << "FUOCO!" <<endl;

  system("PAUSE");
  return 0;
}
```

Dammi il valore da cui partire > 8
8, 7, 6, 5, 4, 3, 2, 1, FUOCO!

Quando il programma inizia viene chiesto all'utente il numero da cui partire con il conto alla rovescia. Quando inizia il ciclo **while**, se il valore introdotto soddisfa la condizione **n>0** (ossia se **n** è maggiore di **0**), il blocco di istruzioni viene ripetuto un numero indefinito di volte fintanto che la condizione **(n>0)** rimane vera.

L'esecuzione del programma precedente si può descrivere con la seguente procedura: cominciando da **main** :

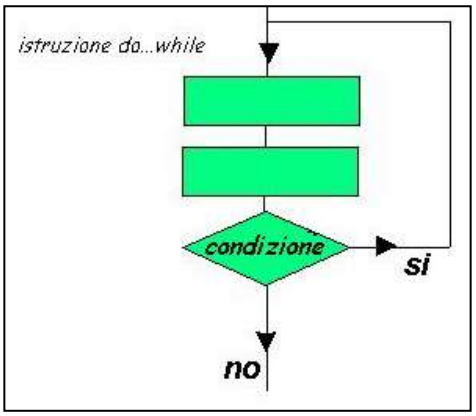
- **1.** L'utente assegna un valore ad **n** .
- **2.** L'istruzione while controlla se **(n>0)** . A questo punto ci sono due possibilità:
  - **true:** esegue il blocco di istruzioni ***Istruzione*** (al punto **3**)
  - **false:** salta il blocco di istruzioni ***Istruzione***. Il programma continua al punto **5** .
- **3.** Esegue il blocco di istruzioni ***Istruzione***:
  cout << n << ", ";
  --n;
  (stampa **n** sullo schermo e diminuisce **n** di 1).
- **4.** Fine del blocco di istruzioni ***Istruzione*** . Ritorna automaticamente al punto **2.**
- **5.** Continua il programma dopo il blocco di istruzioni ***Istruzione***: stampa **FUOCO!** e termina il programma.

Occorre assicurarsi che il ciclo ad un certo punto termini. Per questo occorre prevedere, all'interno del blocco di istruzioni da ripetere, qualche metodo per fare in modo che prima o poi la condizione del ciclo diventi falsa. In caso contrario il ciclo continua indefinitamente. Nel nostro caso abbiamo inserito l'istruzione **--n;** che ci assicura che la condizione diventerà sicuramente falsa dopo un certo numero di ripetizioni: e precisamente quando **n** diventa **0** , ossia quando termina il nostro conto alla rovescia.

Naturalmente, siccome il blocco da ripetere è molto semplice, il calcolatore esegue il programma pressoché istantaneamente, senza apprezzabile intervallo tra un numero e il successivo del conto alla rovescia.

2) Il ciclo *do-while*.

Formato: **do *Istruzione* while (*Condizione*);**



Esso funziona esattamente come il ciclo **while** tranne il fatto che ***Condizion e***viene controllata dopo l'esecuzione di ***Istruzione*** invece che prima di eseguirla.Pertanto con il ciclo *do-while* l'istruzione o blocco di istruzioni ***Istruzione*** viene eseguita sempre almeno una volta, anche se la condizione ***Condizione*** non dovesse risultare mai vera. Ad esempio, il seguente programma ripete ogni numero che l'utente inserisce finché non viene inserito **0**.

```
// ripetitore di numeri
#include <iostream>

using namespace std;
int main ()
{
  unsigned long n;
    do

  {
   cout << "Dammi un numero (0 per finire):
";
   cin >> n;
   cout << "Mi hai dato: " << n << "\n";
   }

   while (n != 0);

system("PAUSE");
  return 0;
}
```

Dammi un numero (0 per finire): 12345
Mi hai dato: 12345
Dammi un numero (0 per finire): 160277
Mi hai dato: 160277
Dammi un numero (0 per finire): 0
Mi hai dato: 0

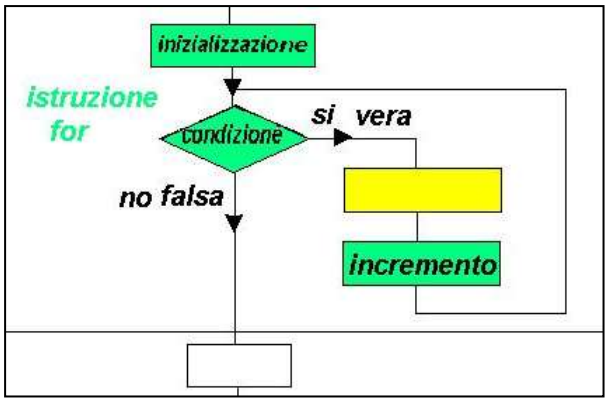
Il ciclo *do-while* si usa quando la condizione che deve terminare il ciclo viene determinata all'interno del ciclo stesso, come nel programma precedente in cui la terminazione del ciclo è determinata dall'input dato dall'utente.

Se l'utente non introduce mai il valore **0** il ciclo continua all'infinito.

3) Il ciclo *for*.

Il suo formato è:

for (*Inizializzazione*; *Condizione* ; *Incremento*) *Istruzione*



ed il suo scopo è quello di ripetere ***Istruzione*** finché la condizione ***Condizione*** rimane vera, esattamente come il ciclo *while*. A differenza di *while* il ciclo *for* permette di indicare anche una istruzione di inizializzazione ***Inizializzazione*** ed una istruzione di incremento ***Incremento***. Il ciclo *for* è quindi particolarmente adatto ad eseguire delle ripetizioni usando un contatore.

Esso opera in questo modo:

1, viene eseguita l'istruzione ***Inizializzazione***. Generalmente essa è una assegnazione di un valore iniziale al contatore. Essa viene eseguita una sola volta.
2, viene controllata la condizione ***Condizione***, se essa è **true** il ciclo continua (al punto **3**), altrimenti il ciclo termina.
3, viene eseguita l'istruzione (o blocco di istruzioni) ***Istruzione***.
4, infine, viene eseguita l'istruzione ***Incremento*** e si ritorna al punto **2**.

Ecco un esempio di conto alla rovescia realizzato con un ciclo *for*:

// conto alla rovescia usando un ciclo for **10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUOCO!**

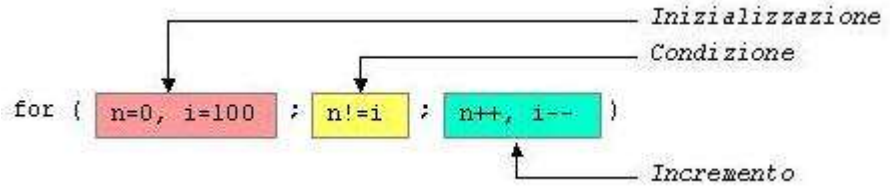
```
#include <iostream>

using namespace std;
int main ()
{
  for (int n=10; n>0; n--)
   {
    cout << n << ", ";
    }
  cout << "FUOCO!";

system("PAUSE");
  return 0;
}
```

I campi ***Inizializzazione*** e ***Incremento*** sono opzionali. Essi si possono quindi omettere ma non si può omettere il punto e virgola che li separa da ***Condizione*** . Possiamo quindi scrivere **for (;n<10;)** se non vogliami indicare né ***Inizializzazione*** né ***Incremento*** oppure **for (;n<10;n++)** se vogliamo includere il campo ***Incremento*** ma non ***Inizializzazione*** .

Possiamo inoltre usare l'operatore virgola (,) per specificare più di una azione in uno qualsiasi dei campi di *for*. L'operatore virgola è un separatore di istruzioni che serve a separare più istruzioni dove ne è richiesta una soltanto. Ad esempio, supponiamo di voler inizializzare più di una variabile per il nostro ciclo:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
  // qualcosa...
}
```

Questo ciclo viene ripetuto 50 volte sempre che né **n** né **i** vengano modificate all'interno del ciclo:



n inizia da **0** e **i** da **100** e la condizione è **n!=i** (ossia che **n** sia diverso da **i**). Siccome **n** viene aumentato di uno ed **i** viene diminuito di uno ad ogni iterazione, la condizione diventerà falsa esattamente dopo **50** iterazioni, quando sia **n** che **i** diventano uguali a **50**.

4) Biforcazioni di controllo e salti.

a) L'istruzione *break*.

Possiamo uscire da un ciclo anche senza che la condizione sia soddisfatta usando una istruzione *break*. Essa si può usare per uscire da un ciclo infinito oppure per forzare la terminazione di un ciclo in presenza di qualche anomalia. Possiamo, ad esempio, interrompere il nostro conto alla rovescia prima che esso termini in modo naturale (a causa di un guasto al motore):

```
// esempio di interruzione di un ciclo
#include <iostream>

using namespace std;
int main ()
{
  int n;
  for (n=10; n>0; n--)

   {
    cout << n << ", ";
    if (n==3)
    {
      cout << "conto alla rovescia interrotto!";
      break;
    }
  }

system("PAUSE");
  return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, conto alla rovescia interrotto!

b) La funzione *exit* .

La funzione *exit* è definita nella libreria standard (stdlib.h).

Essa termina l'esecuzione del programma ritornando un *codice di terminazione* intero. La sua forma è:

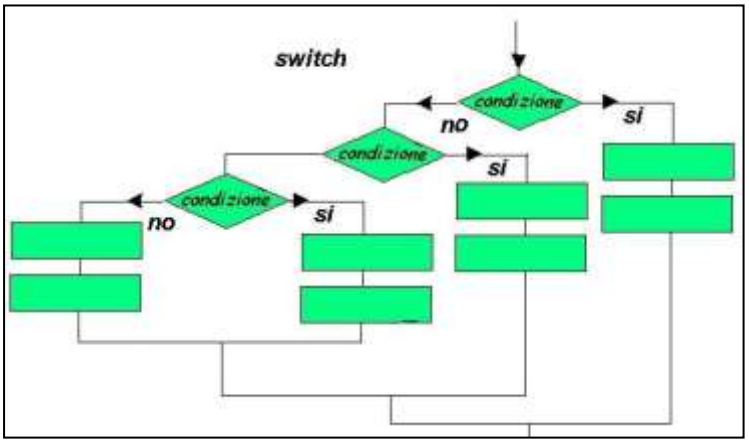
exit(codice) ;

dove **codice** è un valore intero che viene ritornato al sistema operativo. Per convenzione il valore **0** di **cod** significa che il programma è terminato correttamente mentre un valore diverso da zero indica che è stato riscontrato un errore di esecuzione.

c) L'istruzione di scelta: *switch*.

La sintassi dell'istruzione switch è un pò particolare. Il suo scopo è quello di confrontare il valore di una espressione con un certo numero di valori costanti ed eseguire il blocco di istruzioni associato al valore dell'espressione. La sua forma è:

```
switch (Espressione) {
  case Costante1:
    blocco di istruzioni 1
    break;
  case Costante2:
    blocco di istruzioni 2
    break;
  .
  .
  default:
    blocco di istruzioni di default
  }
```



Funziona così: *switch* valuta l'espressione *Espressione* e controlla se essa è uguale a *Costante1*, se lo è esegue il *blocco di istruzioni 1* fino a quando arriva all'istruzione **break** che fa uscire dall'istruzione *switch*. Se *Espressione* non è uguale a *Costante1* essa viene confrontata con *Costante2*. Se è uguale a *Costante2* viene eseguito il *blocco di istruzioni 2* fino all'istruzione **break** . Infine, se l'espressione non risulta uguale a nessuna delle costanti elencate viene eseguito il *blocco di istruzioni di default* se presente (la sezione **default:** è infatti opzionale). *Attenzione*: se si dimentica di inserire l'istruzione **break** alla fine dei blocchi di istruzioni l'esecuzione continua controllando anche le successive costanti ed infine eseguendo il blocco di istruzioni di default.

I seguenti frammenti di codice sono equivalenti:

Esempio *switch*

```
switch (x) {
  case 1:
    cout << "x e' 1";
    break;
  case 2:
    cout << "x e' 2";
    break;
  default:
    cout << "valore di x ignoto";
  }
```

Equivalente *if-else*

```
if (x == 1) {
  cout << "x e' 1";
  }
else if (x == 2) {
  cout << "x e' 2";
  }
else {
  cout << "valore di x ignoto";
  }
```

Attenzione : Se si dimentica di inserire l'istruzione **break** alla fine dei blocchi di istruzioni l'esecuzione continua controllando anche le successive costanti ed infine eseguendo il blocco di istruzioni di default. Questo può essere utile nel caso in cui vogliamo eseguire lo stesso blocco di istruzioni per diversi valori dell'espressione come nell'esempio seguente:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x e' 1, 2 o 3";
    break;
  default:
    cout << "x non e' 1, 2 o 3";
  }
```

Notare che **switch** si può usare soltanto per confrontare una espressione con delle costanti. Non si possono usare variabili o espressioni o intervalli di valori (**case (n*2):** e **case (1..3):** non vanno bene).

Se occorre controllare intervalli di valori o valori non costanti bisogna usare una sequenza di frasi **if** ed **else if** .

Funzioni (I).

L'uso di funzioni permette di strutturare il programma in modo modulare, sfruttando tutte le potenzialità fornite dalla programmazione strutturata fornite dal C++.

Una *funzione* è un blocco di istruzioni con un nome che viene eseguito in ogni punto del programma in cui viene richiamata la funzione usando il nome. Essa si dichiara nel modo seguente:

Tipo Nome **(** ***Argomento1*****,** ***Argomento2*** **, ...)** ***Istruzione***

dove:

- ***Tipo*** è il tipo del valore ritornato dalla funzione.
- ***Nome*** è il nome con cui possiamo richiamare la funzione.
- ***Argomento*** (possiamo indicarne quanti ne vogliamo, anche nessuno). Un argomento è costituito da un nome di tipo seguito da un identificatore (ad esempio **int x**), esattamente come in una dichiarazione di variabile; ed infatti, all'interno della funzione, un argomento si comporta come una variabile (locale). Gli argomenti permettono di passare dei parametri quando la funzione viene richiamata. I parametri sono separati da virgole.
- ***Istruzione*** è il corpo della funzione: un blocco di istruzioni racchiuse tra parentesi graffe **{}** .

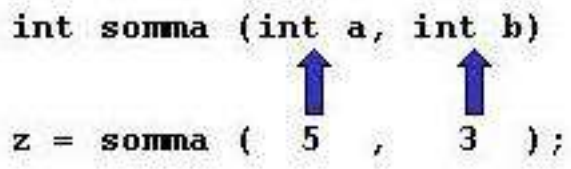
Ecco il primo esempio di funzione:

Il risultato e' 8

```
// esempio di funzione
#include <iostream>
using namespace std;
int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
system("PAUSE");
  return 0;
}
```

Per esaminare questo codice occorre ricordare quello che abbiamo detto all'inizio del corso: un programma C++ inizia sempre dalla funzione **main**. Iniziamo quindi dalla funzione **main** .

La funzione **main** inizia con la dichiarazione della variabile **z** di tipo **int**. Subito dopo vi è una chiamata alla funzione **somma**. Osserviamo la somiglianza della chiamata alla funzione con l'intestazione della dichiarazione della funzione che si trova un po' più sopra:



Vi è una chiara corrispondenza: nella funzione **main** abbiamo richiamato la funzione **somma** passando, come parametri, i due valori **5** e **3** che corrispondono agli argomenti **int a** ed **int b** nella dichiarazione della funzione **somma** .

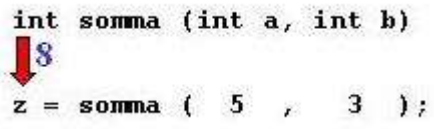
Quando la funzione **somma** viene richiamata dal **main**, il controllo passa dalla funzione **main** alla funzione **somma**. I valori **5** e **3** passati come parametri vengono copiati nelle due variabili **int a** ed **int b** locali alla funzione **somma** .

La funzione somma dichiara una nuova variabile (**int r;**), e quindi, con l'istruzione **r=a+b;**, assegna ad **r** il risultato di **a** più **b** . Siccome i valori passati come parametri sono **5** per **a** e **3** per **b**, il risultato è **8** .

L'istruzione:

return r;

infine termina la funzione **somma** e ritorna alla funzione che l'aveva richiamata (la funzione **main**) riprendendo l'esecuzione dal punto in cui era stata interrotta con la chiamata **somma(5,3)** . L'istruzione **return** ha come argomento la variabile **r** (**return r;**), che al momento dell'esecuzione della **return** ha valore **8**; di conseguenza **8** è il valore ritornato dalla funzione.



Il valore ritornato con l'istruzione **return** è il valore che viene attribuito alla funzione quando essa viene valutata. Ed è proprio tale valore (ossia **8**) che viene assegnato alla variabile **z**.

La riga seguente di **main** è:

cout << "Il risultato e' " << z;

che, come sappiamo, stampa il risultato sullo schermo.

Scopo delle variabili

Il *campo di validità* (scopo) delle variabili dichiarate in una funzione o in un blocco di istruzioni è limitato alla funzione stessa e al blocco di istruzioni e quindi tale variabile non può essere usata al di fuori di tale ambito. Nell'esempio precedente non sarebbe possibile usare le variabili **a** , **b** ed **r** nella funzione **main** in quanto esse sono locali alla funzione **somma**. Analogamente non sarebbe possibile usare la variabile **z** direttamente nella funzione **somma** in quanto essa è locale alla funzione **main**.

Pertanto lo scopo delle variabili locali è limitato al livello di annidamento in cui esse sono dichiarate. Si possono anche dichiarare delle variabili globali che sono visibili in qualunque punto del programma, dentro o fuori a qualsiasi funzione. Per questo occorre dichiarare le variabili globali al di fuori di ogni funzione o blocco, ossia direttamente nel corpo del programma.

Ed ecco un altro esempio sulle funzioni:

```
// esempio di funzioni
#include <iostream>

using namespace std;
int sottrazione (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = sottrazione (7,2);
  cout << "Il primo risultato e' " << z << '\n';
  cout << "Il secondo risultato e' " << sottrazione
```

```
Il primo risultato e' 5
Il secondo risultato e' 5
Il terzo risultato e' 2
Il quarto risultato e' 8
```

```
(7,2) << '\n';
  cout << "Il terzo risultato e' " << sottrazione (x,y)
<< '\n';
  z = 4 + sottrazione (x+2,y);
  cout << "Il quarto risultato e' " << z << '\n';
  return 0;
}
```

In questo caso abbiamo creato la funzione **sottrazione** . L'unica cosa che fa questa funzione è sottrarre i due valori passati come parametro e restituire il risultato.

Se esaminiamo la funzione **main** vediamo che vengono effettuate diverse chiamate alla funzione **sottrazione** . Abbiamo usato diversi modi per chiamare la funzione per illustrare cosa succede quando una funzione viene chiamata.

Il funzionamento di una semplice funzione come sottrazione si può descrivere dicendo che l'effetto di una chiamata a tale funzione è lo stesso che si ottiene sostituendo la chiamata con il suo risultato. Ad esempio, nel primo caso:

```
z = sottrazione (7,2);
cout << "Il primo risultato e' " << z;
```

se sostituiamo la chiamata di funzione con il suo risultato (cioè con **5**), otteniamo:

```
z = 5;
cout << "Il primo risultato e' " << z;
```

Analogamente

```
cout << "Il secondo risultato e' " << sottrazione (7,2);
```

ha lo stesso effetto della chiamata precedente, ma questa volta la chiamata a **sottrazione** viene usata direttamente come argomento dell'operatore di inserimenti (<<) nel flusso **cout** . Sostituendo la chiamata con il suo risultato si ottiene:

```
cout << "Il secondo risultato e' " << 5;
```

in quanto **5** è il risultato di **sottrazione (7,2)** .

Nel caso:

```
cout << "Il terzo risultato e' " << sottrazione (x,y);
```

l'unica novità è che i parametri della chiamata a **sottrazione** sono delle variabili invece che delle costanti. In questo caso i valori passati alla funzione sono i valori delle variabili **x** ed **y** , che sono appunto **5** e **3** ed il risultato che si ottiene è **2**.

Nel quarto caso

```
z = 4 + sottrazione (x+2,y);
```

il primo dei due parametri è una espressione **x+2** ed è il suo valore **7** che viene passato come parametro alla funzione. Il risultato è in questo caso **4** che sostituito al posto della chiamata di funzione fornisce:

```
z = 4 + 4;
```

Funzioni senza risultato. L'uso di *void* .

Ricordiamo la sintassi di una dichiarazione di funzione:

Tipo Nome* (*Argomento1*, *Argomento2* , ...) *Istruzione

si vede che è necessario che essa inizi con un nome di **Tipo** , ossia con il tipo di dato che deve essere ritornato dalla funzione con l'istruzione **return**. E se non vogliamo ritornare alcun valore?

Supponiamo di voler scrivere una funzione che deve soltanto scrivere qualcosa sullo schermo. Non ci serve che

essa ritorni un valore e neppure abbiamo bisogno di passargli dei parametri. Allo scopo il C fornisce un particolare tipo **void**. Osserviamo il seguente esempio:

```
// esempio di funzione void                    Sono una funzione!
#include <iostream>

using namespace std
void stampa (void)
{
  cout << "Sono una funzione!";
}

int main ()
{
  stampa ();
system("PAUSE");
  return 0;
}
```

In C++ l'indicazione di **void** come tipo del risultato o come parametro si può omettere scrivendo semplicemente **stampa ()**. L'uso esplicito di **void** è comunque consigliato per indicare chiaramente che non è richiesto un risultato e/o non sono richiesti parametri (e che quindi non ci si è semplicemente dimenticati di indicarli).

Dobbiamo ricordare che una chiamata di funzione è costituita dal nome della funzione seguita dai parametri racchiusi tra parentesi. Il fatto che non ci siano argomenti non ci esime dal dover scrivere la coppia di parentesi. Di conseguenza la chiamata alla funzione **stampa** è:

stampa ();

il che indica chiaramente che si tratta di una chiamata di funzione e non del nome di una variabile o di qualcosa d'altro.

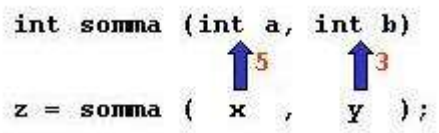
Funzioni (II).

Parametri passati *per valore* e *per riferimento*.

Negli esempi di funzioni visti finora i parametri venivano passati *per valore*. Questo significa che quando viene chiamata una funzione quello che viene passato alla funzione è il valore dei parametri (siano essi delle costanti o delle variabili o delle espressioni). In particolare, se il parametro è una variabile viene passato alla funzione il valore della variabile ma non la variabile stessa. Supponiamo, ad esempio, di richiamare la funzione **somma** nel modo seguente:

```
int x=5, y=3, z;
z = somma ( x , y );
```

In questo caso viene richiamata la funzione **somma** passandogli i valori di **x** ed **y** , ossia **5** e **3** , ma non le variabili stesse:



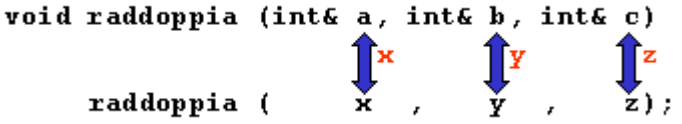
In questo modo, quando la funzione **somma** è chiamata, i valori delle sue variabili **a** e **b** sono **5** e **3** rispettivamente. Una modifica di **a** o **b** all'interno della funzione somma non cambia i valori delle variabili **x** ed **y** esterne ad essa. Questo perché non sono state passate le variabili **x** ed **y** alla funzione **somma** ma soltanto il loro valore .

Ci sono però casi in cui vogliamo modificare dall'interno di una funzione il valore di variabili definite esternamente alla funzione stessa. A questo scopo possiamo usare dei parametri passati *per riferimento*, come nella funzione **raddoppia** dell'esempio seguente:

```
// passaggio di parametri per riferimento                x=2, y=6, z=14
#include <iostream>
using namespace std;
void raddoppia (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  raddoppia (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
system("PAUSE");
  return 0;
}
```

La prima cosa da notare è che nella dichiarazione di **raddoppia** il tipo di ciascun parametro è seguito dal carattere e commerciale (**&**); esso sta ad indicare appunto un passaggio di parametro *per riferimento* invece dell'usuale passaggio per *valore* .

Quando passiamo una variabile per riferimento è la variabile stessa che noi passiamo alla funzione e non soltanto il suo valore. Di conseguenza una modifica del valore del parametro all'interno della funzione modifica il valore della variabile passata come parametro.



In altre parole noi abbiamo associato le variabili locali **a** , **b** e **c** (i *parametri formali* della funzione) alle variabili **x** , **y** e **z** (i *parametri attuali* passati nella chiamata alla funzione) in modo tale che **a** diventa *sinonimo* di **x** , **b** sinonimo di **y** e **c** sinonimo di **z** . Ricordando che una variabile è il nome di una zona di memoria in cui può essere memorizzato un valore (il valore della variabile appunto), dire che **a** e **x** sono sinonimi significa che essi sono nomi diversi per la stessa zona di memoria. Se **a** ed **x** sono sinonimi, una modifica del valore di **a** ha come conseguenza la modifica del valore registrato nella zona di memoria comune ad **a** e **x** e dunque anche il valore di **x** cambia.

Ecco perché l'output del nostro programma, che stampa i valori delle tre variabili **x**, **y** e **z** , mostra che i valori di tali tre variabili sono raddoppiati dopo la chiamata alla funzione **raddoppia** .

Se avessimo dichiarato la funzione raddoppia senza il simbolo *e commerciale* (**&**):

void raddoppia (int a, int b, int c)

non avremmo passato le variabili **x**, **y** e **z** ma soltanto i loro valori e quindi il programma avrebbe stampato i valori di **x** , **y** e **z** non modificati.

Il passaggio di parametri per riferimento permette di scrivere funzioni che calcolano più di un valore. Ad esempio, ecco una funzione che calcola il numero precedente ed il numero successivo del primo parametro che gli viene passato:

```
// calcolo di piu' di un valore                Precedente=99, Successivo=101
#include <iostream>
using namespace std;
void precsucc (int x, int& prec, int& succ)
```

```
{
  prec = x-1;
  succ = x+1;
}

int main ()
{
  int x=100, y, z;
  precsucc (x, y, z);
  cout << "Precedente=" << y << ", Successivo=" << z;
 system("PAUSE");
 return 0;
}
```

Valori di default per i parametri.

Nella dichiarazione di una funzione si possono specificare dei *valori didefault* per i parametri. I valori di default vengono usati nel caso in cui tali parametri vengano omessi nella chiamata di funzione. Ad esempio:

```
// valori di default per i parametri          6
#include <iostream>                           5
using namespace std;
int dividi (int a, int b=2)
{
  int r;
  r=a/b;
  return r;
}

int main ()
{
  cout << dividi (12);
  cout << endl;
  cout << dividi (20,4);
  return 0;
}
```

Nel programma precedente ci sono due chiamaste alla funzione **dividi**. Nella prima:

divide (12)

viene passato un solo argomento mentre la funzione ne richiede due. Siccome il secondo parametro ha valore di default **2** è proprio il valore **2** che viene passato implicitamente come valore del secondo parametro **b**. Quindi il risultato che si ottiene è **6** (**12/2**).

Nella seconda chiamata:

dividi (20,4)

vi sono entrambi i parametri, quindi il valore di default **2** non viene usato ma viene passato il valore **4** come valore del secondo parametro **b**. Quindi il risultato che si ottiene è **5** (**20/4**).

Notare che la corrispondenza tra parametri attuali e parametri formali è posizionale e quindi in una chiamata si possono omettere soltanto gli ultimi parametri. Di conseguenza, in una chiamata di funzione con valori di default per i parametri è possibile omettere tutti i parametri da un certo punto in poi ma non ometterne uno intermedio.

Funzioni sovraccaricate (overloading).

Due funzioni distinte possono avere lo stesso nome purché la lista degli argomenti sia diversa. Questo significa che possiamo dare lo stesso nome a più di una funzione purché esse abbiano un diverso numero di parametri o almeno un parametro di dipo diverso. Ad esempio:

```
// funzione sovraccaricata
#include <iostream>
using namespace std;
int dividi (int a, int b)
{
  return a/b;
}

float dividi (float a, float b)
{
  return a/b;
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << dividi (x,y);
  cout << "\n";
  cout << dividi (n,m);
  cout << "\n";
  return 0;
}
```

```
2
2.5
```

In questo caso abbiamo definito due funzioni con lo stesso nome dividi ma con parametri di tipo diverso: la prima con due parametri di tipo int, la seconda con due parametri di tipo float. Il compilatore decide quale delle due debba essere chiamata esaminando il tipo dei parametri attuali forniti nella chiamata.

Nell'esempio le due funzioni hanno lo stesso corpo ma questo non è necessario: funzioni con lo stesso nome possono anche fare cose completamente diverse.

Ricorrenza.

La *ricorrenza* (o *ricorsività*) è la proprietà di una funzione di poter essere richiamata da se' stessa, ossia all'interno del corpo della funzione possono comparire chiamate alla funzione stessa. Questa possibilità risulta particolarmente utile in certe situazioni in cui il valore da calcolare può essere definito per induzione . Ad esempio, il fattoriale di un numero intero **n**:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

si può definire induttivamente nel seguente modo:

- se **n=1** allora **n! = 1**
- se **n>1** allora **n!= n * (n-1)!**

il che suggerisce la seguente funzione ricorsiva:

```
// calcolo del fattoriale
#include <iostream>
using namespace std;
long fattoriale (long a)
{
  if (a > 1)
```

```
Dammi un numero:9
9! = 362880
```

```
  return a * fattoriale (a-1);
 else
  return 1;
}

int main ()
{
 long n;
 cout << "Dammi un numero: ";
 cin >> n;
 cout << n << "!"<< " = " << fattoriale (n);
system("PAUSE");
 return 0;
}
```

Osserviamo che nella funzione **fattoriale** viene ricorsivamente effettuata una chiamata alla funzione **fattoriale** stessa. La chiamata ricorsiva viene però effettuata soltanto se l'argomento è maggiore di **1** , altrimenti la funzione entrerebbe in un *ciclo di ricorsione infinito* , venendo richiamata con argomento **0** e quindi con argomento **-1** , **-2** e così via.

Un'altra osservazione è che la funzione fattoriale ha una limitazione nei valori dell'argomento dovuta al fatto che il fattoriale di un intero cresce molto rapidamente. In pratica, il tipo del risultato (**long**) non permette di memorizzare fattoriali maggiori di **12!**.

Prototipi di funzioni.

Finora abbiamo sempre messo la definizione di una funzione prima della prima occorrenza di una chiamata alla funzione stessa che generalmente appare nella funzione main. Per questa ragione abbiamo dovuto mettere sempre la funzione main alla fine. Se negli esempi precedenti avessimo messo la funzione main per prima avremmo ottenuto una segnalazione di errore. La ragione è che quando viene richiamata una funzione essa deve essere già nota al compilatore.

In realtà il compilatore per poter effettuare una chiamata di funzione ha bisogno di conoscere soltanto il nome della funzione ed il numero e tipo dei suoi parametri (il *prototipo della funzione*) mentre non ha alcun bisogno di conoscerne il corpo. Il C++ permette di dichiarare il prototipo di una funzione in modo tale da renderla nota al compilatore e rimandare in seguito la definizione vera e propria della funzione (comprendente anche il corpo).

La forma di una dichiarazione di prototipo è la seguente :

***tipo nome* (*tipo_parametro1*, *tipo_parametro2*, ...);**

ed è simile alla intestazione di una dichiarazione di funzione eccetto:

- manca la parte *Istruzione*, ossia il blocco di istruzioni racchiuso tra parentesi graffe **{}** che costituisce il corpo della funzione.
- termina con il carattere punto e virgola (;).
- nell'elenco dei parametri basta indicare i tipi degli argomenti anche se è consigliabile mettere anche il nome del parametro benché esso sia opzionale.

Ad esempio:

```
// prototipazione

#include <iostream>
using namespace std;
void dispari (int a);
```

Scrivi un numero (0 per uscire): 9
Il numero è dispari.
Scrivi un numero (0 per uscire): 6
Il numero è pari.
Scrivi un numero (0 per uscire): 1030

```
void pari (int a);
int main ()
{
  int i;
  do {
    cout << "Scrivi un numero: (0 per uscire)";
    cin >> i;
    dispari (i);
  } while (i!=0);
  return 0;
}

void dispari (int a)
{
  if ((a%2)!=0) cout << "Il numero è dispari.\n";
  else pari (a);
}
void pari (int a)
{
  if ((a%2)==0) cout << "Il numero è pari.\n";
  else dispari (a);
}
```

```
Il numero è pari.
Scrivi un numero (0 per uscire): 0
Il numero è pari.
```

Questo esempio non ha una grande utilità: chiunque sarebbe in grado di ottenere lo stesso risultato con un programma molto più semplice. Ma lo scopo dell'esempio è illustrare come funziona la prototipazione. Inoltre, in questo caso la prototipazione di almeno una delle due funzioni **dispari** e **pari** è indispensabile in quanto le due funzioni si richiamano vicendevolmente.

All'inizio del programma compaiono i prototipi delle funzioni **dispari** e **pari**:

```
void dispari (int a);
void pari (int a);
```

che permettono di usare le due funzioni prima che esse siano completamente definite, ad esempio in **main** che adesso può essere messo nella posizione più logica, cioè all'inizio del programma.

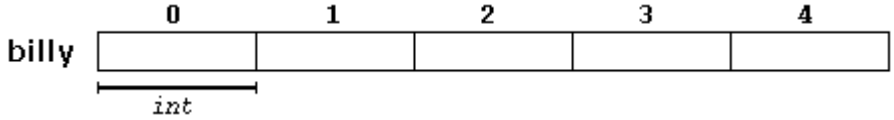
Molti programmatori esperti consigliano di prototipare tutte le funzioni. Questo è particolarmente utile quando un programma contiene molte funzioni o le definizioni delle funzioni sono molto lunghe. In tal caso raccogliere tutti i prototipi nello stesso posto all'inizio facilita la ricerca se non si ricorda come devono essere richiamate (numero e tipo dei parametri).

Array

Gli array sono sequenze di variabili dello stesso tipo che vengono situate consecutivamente nella memoria ed alle quali è possibile accedere usando uno stesso nome (identificatore) a cui viene aggiunto un indice.

Questo significa, ad esempio, che possiamo memorizzare **5** valori di tipo **int** senza bisogno di dichiarare cinque diverse variabili con cinque diversi identificatori. Per fare questo è sufficiente dichiarare un *array* di cinque elementi dello stesso tipo **int** con un solo identificatore.

Ad esempio un array di nome **billy** contenente **5** valori di tipo **int** si può rappresentare nel seguente modo:



in cui ogni cella rappresenta un *elemento* dell'array. Gli elementi sono numerati da **0** a **4** in quanto, in un array, l'indice del primo elemento è sempre **0** (e non **1**).

Come tutte le variabili anche gli array devono essere dichiarati prima di poterli usare. Un esempio di dichiarazione di un array in C++ è:

***tipo nome* [*dimensione*];**

dove ***tipo*** è il tipo degli elementi (**int**, **float** ...) detto anche *tipo base dell'array* , ***nome*** è un *identificatore* e ***dimensione***, che deve essere racchiuso tra parentesi quadre **[]**, è la *dimensione,* ossia il numero di elementi, dell'array.

La dichiarazione dell'array *billy* è:

int billy [5];

ATTENZIONE: Il campo **dimensione** deve essere un valore costante in quanto gli array sono blocchi di memoria di dimensione prefissata ed il compilatore deve conoscere esattamente quanta memoria serve per l'array prima che il programma venga eseguito.

Inizializzazione degli array.

Un array può essere inizializzato in due modi:

- Esplicitamente, al momento della creazione, fornendo le costanti di inizializzazione dei dati.
- Durante l'esecuzione del programma, assegnando o copiando dati nell'array.

Vediamo un esempio di inizializzazione esplicita al momento della creazione:

int billy [5] = { 16, 2, 77, 40, 12071 };

l'array viene inizializzato come segue:

	0	1	2	3	4
billy	16	2	77	40	12071

Il numero di valori usati per l'inizializzazione (quelli posti tra le parentesi grafe **{}**) deve essere esattamente uguale alla dimensione dell'array. In C++ è possibile anche usare la notazione:

int billy [] = { 16, 2, 77, 40, 12071 };

ed in questo caso viene assunto implicitamente come dimensione dell'array il numero di valori della lista di inizializzazione.

Per inizializzare un array durante l'esecuzione del programma occorre accedere, generalmente con un ciclo, ad ogni elemento dell'array stesso ed assegnargli un valore. L'accesso ad un elemento di un array avviene indicando il nome dell'array e, tra parentesi quadre, l'indice corrispondente all'elemento voluto. Così, x[2] indicherà il terzo elemento dell'array x e non il secondo poichè la numerazione degli indici, come abbiamo detto, inizia da zero. Vediamo ora un esempio di inizializzazione eseguita durante l'esecuzione del programma:

```
int numeri_pari[10];
int i;

for(i =0; i < 10; i++)
{
        numeri_pari[i] = i * 2 ;
}
```

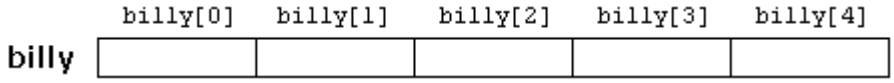
Il programma precedente non fa altro che assegnare all'array i numeri pari da 0 ad 18.

Accesso ai valori di un Array.

In ogni punto del programma in cui un array risulta visibile possiamo accedere individualmente ad uno degli elementi dell'array per leggerlo o modificarlo esattamente come esso fosse una normale variabile. Il formato è il seguente:

***name*[*index*]**

Proseguendo l'esempio dell'array **billy** di **5** elementi di tipo **int**, i nomi mediante i quali possiamo accedere a ciascun elemento dell'array sono quelli indicati sopra le singole celle nella figura seguente:



Ad esempio, se vogliamo memorizzare il valore **75** nel terzo *elemento* di **billy** possiamo usare l'assegnazione:

billy[2] = 75;

oppure, per copiare il valore del terzo elemento nella variabile **a** possiamo usare:

a = billy[2];

Dunque, a tutti gli effetti, **billy[2]** si comporta come una variabile di tipo **int**.

Notiamo che il terzo elemento di **billy** è **billy[2]**, infatti **billy[0]** è il primo, **billy[1]** è il secondo e di conseguenza **billy[2]** è il terzo. Per la stessa ragione l'ultimo elemento è **billy[4]** . Se scriviamo **billy[5]**, noi accediamo al sesto elemento dell'array che non è previsto incorrendo molto probabilmente in un errore di esecuzione.

A questo punto occorre notare i due diversi usi delle parentesi quadre **[ ]** con gli array: nella dichiarazione di un array esse sono usate per indicare la dimensione dell'array mentre in tutti gli altri contesti esse vengono usate per specificare un indice per individuare un particolare elemento dell'array. Ad esempio:

```
int billy[5];       // dichiarazione di un nuovo array di 5 elementi
billy[2] = 75;      // accesso ad un elemento particolare dell'array: quello di indice 2.
```

Altre possibili operazioni con gli array sono:

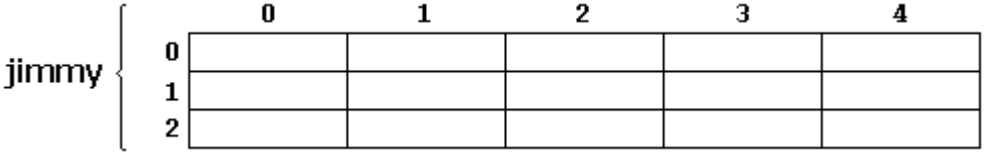
```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

// esempio con gli array **12206**

```
#include <iostream.h>

int billy [] = {16, 2, 77, 40, 12071};
int n, risultato=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    risultato += billy[n];
  }
  cout << risultato;
  return 0;
}
```

Array multidimensionali

Un array multidimensionale si può pensare come un array di array di array di .... Ad esempio, un array bidimensionale si può pensare come una tabella bidimensionale i cui elementi appartengono tutti allo stesso *tipo*.

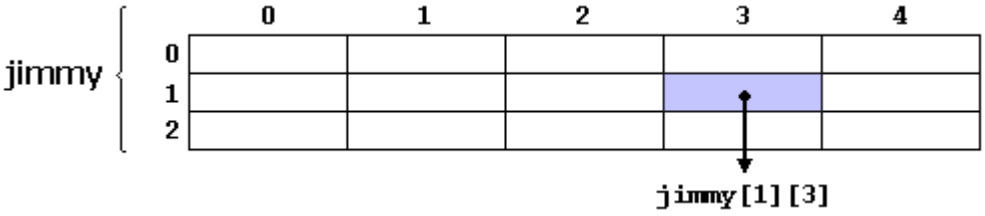


jimmy è un array bidimensionale di 3 per 5 valori di tipo **int**. Esso si può pensare come un array di 5 elementi, ciscuno dei quali è a sua volta un array di 3 elementi (le colonne). Esso si dichiara come segue:

int jimmy [3][5];

e, per riferirsi all'elemento nella seconda riga e nella quarta colonna si usa la notazione:

jimmy[1][3]



(ricordiamo che gli indici degli array iniziano *sempre* con **0**).

Gli *array multidimensionali* possono avere più di due indici (due dimensioni). Possono avere quanti indici vogliamo ma è molto raro che servano più di 3 dimensioni. La memoria necessaria per un array multidimensionale può essere eccessiva. Ad esempio:

char secolo [100][365][24][60][60];

richiede memoria per un valore **char** per ogni secondo contenuto in un secolo, più di 3 miliardi di **char** ! Il che richiede più di 3 *gigabytes* di memoria RAM.

Gli elementi di un array multidimensionale sono memorizzati nella RAM uno di seguito all'altro come per gli array semplici. Potremmo ottenere lo stesso risultato usando un array semplice di dimensione il prodotto delle dimensioni dell'array multidimensionale:

int jimmy [3][5]; è equivalente a
int jimmy [15];

con l'unica differenza che il compilatore gestisce per noi la suddivisione in righe e colonne. Il seguente esempio mostra l'equivalenza delle due soluzioni:

```
// array multidimensionale
#include <iostream.h>

#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE][COLONNE];
int n,m;

int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
// array pseudo-multidimensionale
#include <iostream.h>

#define COLONNE 5
#define RIGHE 3

int jimmy [RIGHE * COLONNE];
int n,m;

int main ()
{
  for (n=0;n<RIGHE;n++)
    for (m=0;m<COLONNE;m++)
    {
      jimmy[n * COLONNE + m]=(n+1)*(m+1);
    }
  return 0;
}
```

entrambi i programmi assegnano i seguenti valori al blocco di memoria riservato per **jimmy**:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Abbiamo usato la definizione di costanti (**#define**) per facilitare future modifiche al programma nel caso, ad esempio, che decidessimo di allargare l'array aggiungendo una riga. Basta in questo caso cambiare la riga:

#define RIGHE 3

con

#define RIGHE 4

e non serve nessun'altra modifica del programma.

Stringhe di Caratteri.

Nei programmi visti finora abbiamo usato soltanto variabili numeriche i cui valori possono essere unicamente dei numeri (ricordiamo che anche i valori di variabili di tipo **char** sono interpretati come numeri interi). Oltre alle variabili numeriche si possono anche dichiarare variabili i cui valori sono stringhe di caratteri. Tali variabili ci permettono di elaborare successioni di caratteri quali parole, frasi, nomi, testi, eccetera. Finora abbiamo usato le stringhe di caratteri come costanti ma non abbiamo ancora visto variabili che possano contenerle.

In C++ non vi è un tipo di variabile *predefinito* in grado di memorizzare delle stringhe di caratteri. Dobbiamo usare degli array di caratteri. In C++ sono comunque predefinite alcune operazioni e funzioni sugli array di caratteri (ereditate dal C) che facilitano il loro uso come variabili di tipo *stringa di caratteri*.

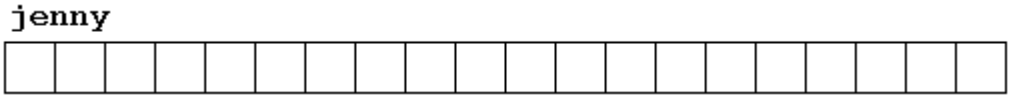
La libreria standard del C++ contiene un file (che si può includere con il comando **#include <string>**) in cui è definito un tipo **string** con il quale l'elaborazione di stringhe risulta molto agevolata.

Vediamo ora il trattamento standard (stile C) delle stringhe come array di caratteri.

Il seguente array:

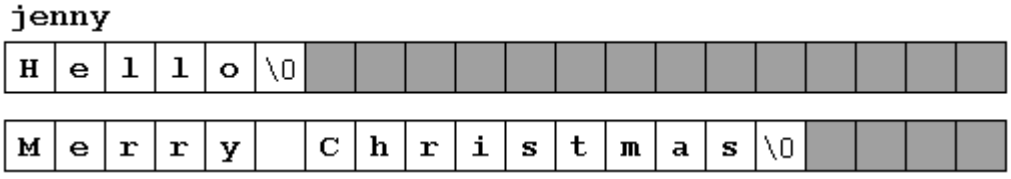
char jenny [20];

può memorizzare una stringa di al più 20 caratteri. Possiamo rappresentarlo come segue:



Naturalmente non è necessario usare tutti e 20 i caratteri dell'array. L'array **jenny** si può usare per memorizzare sia la stringa di 5 caratteri **"Hello"** sia la stringa di 15 caratteri **"Merry Christmas"**. Siccome l'array può contenere stringhe più corte della sua dimensione occorre prevedere una indicazione del punto in cui termina la stringa. Per convenzione tale punto viene indicato dal carattere nullo che si può scrivere sia **0** sia '**\0**'.

Ad esempio:



Osserviamo che dopo il contenuto effettivo della stringa viene aggiunto un carattere nullo ('**\0**') per indicare la fine della stringa. Pertanto l'array jenny può contenere al più stringhe di 19 caratteri.

Inizializzazione delle stringhe

Per inizializzare una stringa di caratteri si può usare la stessa notazione usata per gli array:

char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };

Abbiamo così inizializzato una stringa (array) di 6 valori di tipo **char**: la parola **Hello** più il carattere nullo '**\0**'.

Possiamo anche inizializzare un array di caratteri usando una stringa costante. Abbiamo già visto delle stringhe costanti quali ad esempio:

"Risultato: "

e le abbiamo usate in istruzioni di stampa. Esse si rappresentano racchiudendo la sequenza di caratteri tra doppi apici ("). Alle stringhe costanti viene sempre aggiunto implicitamente un carattere nullo finale '**\0**'.

Possiamo quindi inizializzare indifferentemente la variabile **mystring** in uno dei modi seguenti:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [] = "Hello";
```

In entrambi i casi la dimensione dell'array **mystring** è di 6 elementi di tipo **char**: i 5 caratteri di **Hello** e il carattere nullo finale ('\0').

Attenzione: una stringa costante può essere usata per dare un valore iniziale ad una variabile di tipo array di **char** soltanto al momento della dichiarazione dell'array, ossia in fase di inizializzazione. Assegnazione quali:

```
mystring = "Hello";
mystring[] = "Hello";
```

non sono permesse, come non è permesso:

```
mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Assegnazione di valori alle stringhe

Siccome in una assegnazione l'*lvalore* può essere soltanto un elemento di un array e non l'intero array, per assegnare una stringa di caratteri ad un array di **char** dobbiamo scrivere:

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

Siccome questo non è molto pratico la libreria standard **cstring** (che si può includere con **#include <string.h>**) contiene la definizione di un certo numero di funzioni quali **strcpy** (**str** ing **c**o **py**) che si può richiamare nel seguente modo:

strcpy (*string1***,** *string2***);**

L'effetto è copiare il contenuto di *string2* in *string1*. *string2* può essere sia un array sia una stringa costante, il che ci permette di assegnare la stringa costante **"Hello"** al'array di caratteri **mystring** usando la seguente notazione:

```
strcpy (mystring, "Hello");
```

Vediamo un esempio:

```
// assegnazione a stringhe
#include <iostream.h>
#include <string.h>
int main ()
{
  char stMyName [20];
  strcpy (stMyName,"J. Soulie");
  cout << stMyName;
  return 0;
}
```

```
J. Soulie
```

Un altro modo per assegnare un valore ad un array di caratteri è quello di usare direttamente il flusso di input **cin**. Nella libreria iostream è infatti definita una funzione **getline** il cui prototipo è:

cin.getline (char *buffer* **[], int** *length*, **char** *delimiter* = ' **\n');**

dove ***buffer*** è l'array di caratteri in cui memorizzare l'input, ***length*** è la dimensione dell'array stesso e ***delimiter*** è il carattere usato per indicare la fine dell'input e per il quale è previsto il carattere *nuova linea* ('**\n**') come valore di default.

(in C usando la libreria standard <stdio.h> l'istruzione è gets)

Ecco un esempio di uso di **cin.getline** :

```
// uso di cin.getline
#include <iostream.h>

int main ()
{
  char buff [100];
  cout << "Come ti chiami? ";
  cin.getline (buff,100);
  cout << "Salve " << buff << ".\n";
  cout << "La tua squadra preferita? ";
  cin.getline (buff,100);
  cout << "L'" << buff << " piace anche a me.\n";
  return 0;
}
```

```
Come ti chiami? Juan
Salve Juan.
La tua squadra preferita? Inter
L'Inter piace anche a me
```

Si può anche usare l'operatore di estrazione (>>) per leggere delle stringhe da **cin** :

cin >> buff;

che funziona ma con le seguenti limitazioni che **cin.getline** non ha:

- si possono leggere soltanto parole e non intere frasi in quanto l'operatore di estrazione usa come delimitatore qualsiasi occorrenza di un carattere invisibile (spazio, tabulazione, nuova linea, ritorno carrello).
- non si può specificare la dimensione dell'array il che rende instabile il programma nel caso in cui l'input sia una parola più lunga della dimensione dell'array.

Le seguenti sono altre funzioni che operano su stringhe e che sono definite nella libreria **cstring** (**string.h**):

strcat:

char* strcat (char* *dest* **, const char*** *src* **);**

Aggiunge (appende) la stringa *src* alla fine della stringa *dest*. Ritorna *dest*.

strcmp:

int strcmp (const char* *str1*, **const char*** *str2* **);**

Confronta le stringhe *str1* ed *str2* . Ritorna **0** se sono uguali.

strcpy:

char* strcpy (char* *dest*, **const char*** *src* **);**

Copia il contenuto di *src* in *dest*. Ritorna *dest*.

strlen:

size_t strlen (const char* *str*);

Ritorna la lunghezza di *str*.

NOTA: **char*** ha lo stesso significato di **char[]**