**ABC++**

****



Copyright © Ing. Daniele Corti 2013

[www.ingdanielecorti.it](http://www.ingdanielecorti.it)

Tutti i diritti sono riservati a norma di legge e a norma delle convenzioni internazionali.

v.4.2 – 13/10/2019

**PREREQUISITI**

* Risolvere problemi strutturati con gli algoritmi attraverso i flowchar.
* Riconoscere le sequenze, le selezioni e le iterazioni.

**OBIETTIVI**

* Saper scrivere un programma in C++ sintatticamente corretto.
* Saper interpretare uno script in C++.
* Risolvere problemi di natura scientifica attraverso il linguaggio di programmazione C++.

**ARGOMENTI TRATTATI (num di pag da aggiornare)**

Introduzione …………………………………………………………………………………... Pag.04

L’operatore d’inserimento COUT …………………………………………………………….. Pag.06

Gli identificatori ………………………………………………………………………………. Pag.09

Le variabili ……………………………………………………………………………………. Pag.10

I tipi standard del C++ ………………………………………………………………………… Pag.11

I tipi derivati …………………………………………………………………………………... Pag.13

I caratteri ……………………………………………………………………………………… Pag.13

Approfondimento – I/O in C – printf – scanf ………………………………………………… Pag.14

Gli interi ………………………………………………………………………………………. Pag.16

Numeri in virgola mobile ……………………………………………………………………… Pag.16

Dichiarazione di variabili ……………………………………………………………………... Pag.17

Blocchi di istruzione ………………………………………………………………………….. Pag.18

Le costanti …………………………………………………………………………………….. Pag.19

La direttiva #define …………………………….……………………………………… Pag.19

Il modificatore const …………………………………………………………………... Pag.20

Operatori booleani …………………………………………………………………………….. Pag.21

Operatori aritmetici …………………………………………………………………………… Pag.23

Operatori di assegnamento ……………………………………………………………………. Pag.24

Operatori di relazione …………………………………………………………………………. Pag.25

Precedenza tra operatori ………………………………………………………………………. Pag.25

L’operazione di estrazione tramite CIN ………………………………………………………. Pag.27

L’operatore d’incremento ………………………………………………………………...…… Pag.28

Controllo del flusso …………………………………………………………………………… Pag.30

La sequenza …………………………………………………………………………… Pag.30

La selezione ………………………………………………………………………….... Pag.31

Il costrutto IF … ELSE ………………………………………………………………... Pag.31

Istruzioni IF … ELSE nidificate ………………………………………………. Pag.34

Le istruzioni SWITCH ………………………………………………………… Pag.35

Le iterazioni …………………………………………………………………………… Pag.38

Il ciclo WHILE ……………………………………………………………...… Pag.38

Il ciclo DO … WHILE ……………………………………………………….... Pag.40

Il ciclo FOR …………………………………………………………………… Pag.44

Gli array ………………………………………………………………..……………………… Pag.47

Gli array a una dimensione ……………………………………………………………. Pag.47

Dichiarazione di un array ……………………………………………………………… Pag.49

Inizializzazione di un vettore ………………………………………………………….. Pag.53

Esempio ……………………………………………………………………………….. Pag.54

Stringhe ……………………………………………………………………………….. Pag.58

La libreria string ………………………………………………………………………. Pag.59

Gestione delle parole separate dallo spazio ……………………………………………. Pag.62

Array paralleli …………………………………………………………………………. Pag.62

Array multimensionali ………………………………………………………………… Pag.65

Le matrici ……………………………………………………………………………… Pag.65

Procedure – funzioni ………………………………………………………………………….. Pag.70

Approfondimento Trasmissione dei parametri tramite l’area stack …………………… Pag.82

Approfondimento Ricorsione delle funzioni ………………………………………..… Pag.83

Approfondimento Visibilità e tempo di vita di una variabile …………………………. Pag.86

Approfondimento Invocazione di una funzione ………………………………………. Pag.89

Approfondimento Rispettare l’ordine nel passaggio dei parametri …………………… Pag.91

Esercizi di base ………………………………………………………………………... Pag.93

Passaggio dei parametri per valore ……………………………………………………. Pag.93

Passaggio dei parametri per riferimento ………………………………………………. Pag.94

Approfondimento Passaggio dei parametri per indirizzo ……………………………… Pag.97

Analisi di una funzione ………………………………………………………………... Pag.98

Esercizi ………………………………………………………………………………... Pag.98

# INTRODUZIONE

Come è facile intuire, il linguaggio C++ nasce come estensione del linguaggio C. In particolare, il C++ conserva tutti i punti di forza del C, come la potenza e la flessibilità di gestione dell’interfaccia hardware e software, la possibilità di programmare a basso livello e l’efficienza, l’economia e le espressioni, tipiche del C.

Il notevole valore aggiunto del C++ è la [programmazione orientata agli oggetti](http://programmazione.html.it/guide/leggi/38/guida-programmazione-orientata-agli-oggetti/), che rende questo linguaggio una piattaforma ideale per realizzare progetti di grosse dimensioni favorendo l’astrazione dei problemi e l’uso dei modelli di programmazione più moderni.

Il C++ fonde, quindi, i costrutti tipici dei linguaggi procedurali standard, familiari a molti programmatori, con il modello di programmazione orientata agli oggetti, che può essere pienamente sfruttato per produrre soluzioni completamente orientate agli oggetti di un determinato problema.

In pratica, una applicazione C++ riflette questa dualità incorporando sia il modello di programmazione procedurale sia il modello OOP (Object Oriented Programming).

Questa dispensa ABC++ si rivolge sia a chi si affaccia per la prima volta allo straordinario mondo della programmazione, sia a chi ha desiderio di rispolverare qualche concetto non troppo chiaro.

L’obiettivo, in ogni caso, non è tanto quello di fornire una descrizione approfondita di tutte le potenzialità della programmazione ad oggetti, che demandiamo ad altre guide, ma più semplicemente, quello di fornire una panoramica sul C++ che permetta rapidamente di essere pronti a scrivere applicazioni.

# LA STORIA DEL C++

Non dovrebbe sorprendere più di tanto il fatto che il C++ abbia un’origine simile al C. Lo sviluppo del linguaggio C++ [[1]](#footnote-1)all’inizio degli anni Ottanta è dovuto a **Bjarne Stroustrup** dei laboratori Bell (Stroustrup ammette che il nome di questo nuovo linguaggio è dovuto a Rick Mascitti).

Stroustrup, nato nel 1950, è un programmatore e informatico danese a capo dell’*AT&T[[2]](#footnote-2) Lab’s Large-scale Programming Research department*, il dipartimento di ricerca e programmazione su larga scala della AT&T, la più grande compagnia telefonica statunitensi con sede a San Antonio, Texas. In particolare tra i centri di ricerca AT&T spiccano i Bell Laboratories, dove sono stati sviluppati il transistor, i MOSFET, i CCD (DAC in italiano, dispositivi di accoppiamento di carica), il linguaggio C e C++, il sistema operativo Unix, WaveLAN la prima rete wireless e tanto altro.

Stroustrup ha inoltre scritto quello che molti considerano il testo standard di introduzione al linguaggio, "The C++ Programming Language" (titolo italiano: "C++ Linguaggio, libreria standard, principi di programmazione"), giunto alla terza edizione.



Bjarne Stroustrup

Originariamente il C++ era stato sviluppato per risolvere alcune simulazioni molto rigorose e guidate da eventi; per questo tipo di applicazione la scelta della massima efficienza precludeva l’impiego di altri linguaggi.

Il linguaggio C++ venne utilizzato all’esterno del gruppo di sviluppo di Stroustrup nel 1983 e, fino all’estate del 1987, il linguaggio fu soggetto a una naturale evoluzione.

Uno degli scopi principali del C++ era quello di mantenere piena compatibilità con il C. L’idea era quella di conservare l’integrità di molte librerie C e l’uso degli strumenti sviluppati per il C. Grazie all’alto livello di successo nel raggiungimento di questo obiettivo, molti programmatori trovano la transizione al linguaggio C++ molto più semplice rispetto alla transizione da altri linguaggi (ad esempio il FORtrAN) al C.

Il C++ consente lo sviluppo di software su larga scala. Grazie a un maggiore rigore sul controllo dei tipi, molti degli effetti collaterali tipici del C, divengono impossibili in C++.

Il miglioramento più significativo del linguaggio C++ è il supporto della programmazione orientata agli oggetti (Object Oriented Programming: OOP), Per sfruttare tutti i benefici introdotti dal C++ occorre cambiare approccio nella soluzione dei problemi. Ad esempio, occorre identificare gli oggetti e le operazioni ad essi associate e costruire tutte le classi e le sottoclassi necessarie.

# L’OPERATORE D’INSERIMENTO COUT

# Vediamo come primo esempio un programma molto semplice il cui scopo è scrivere sul monitor la frase "Questo e' il mio primo programma in C++".

|  |  |
| --- | --- |
| Num | Codice |
| 123456789101112131415161718 | /\* Paolo Rossi 2F – Liceo Scientifico A.Tosi – 2013/2014 Il mio primo programma in C++\*/#include <iostream>#include<stdlib.h>using namespace std;int main(int argc, char\*\* argv){ // Il programma inizia da quì char carattere; char num; cout<<"Questo e' il mio primo programma in C++"; cout<<endl; system("PAUSE"); return(0);} |

# Osserviamo che una caratteristica tipica degli editor di codice è la diversa colorazione del testo a seconda del relativo significato sintattico: syntax highlighting. Questa proprietà è sicuramente un vantaggio dal punto di vista di lettura e interpretazione del codice digitato.

# Analizziamo il codice:

# Alla riga 14 troviamo la prima vera e propria istruzione del nostro programma, l’istruzione cout<<"Questo e' il mio primo programma in C++"; che permette la stampa a video del messaggio "Questo e' il mio primo programma in C++". Questa istruzione appartiene al namespace std e viene fornito dall’inclusione dell’header <iostream> come si vedrà in seguito. Analizziamo tale istruzione:

# cout, è la parola chiave che significa “consolle output” e di default è il monitor la periferica predefinita di uscita.

# La frase Questo e' il mio primo programma in C++ scritta fra le virgolette è il messaggio, una sequenza di caratteri ovvero una stringa.

# L’operatore d’inserimento << è utilizzato per inserire/inviare delle stringhe alla console output. Spesso si parla di flusso per indicare che i dati fluiscono da o verso una periferica. I dati subiscono una conversione prima di essere inviati in streming, e quindi, l’operazione è cumulabile, ovvero si possono inviare più dati allo stesso stream: cout << dato1 << dato2 << daton;. Ne riparleremo più avanti.

# Il punto e virgola per indicare la fine dell’istruzione.

# Dalla riga 1 alla riga 4 troviamo un commento. Un commento è un testo che non sarà preso in considerazione dal compilatore, ma che è utile come promemoria per i programmatori che leggeranno quanto scritto. Saper commentare efficacemente il proprio lavoro è un’attività importante da parte di un buon programmatore. Osserviamo che il commento deve essere scritto, anche su più righe, fra la sequenza /\* e \*/.

# Alla riga 11 troviamo un secondo modo di scrivere i commenti, ereditato dal linguaggio C, su una sola riga. Il commento inizia dopo la sequenza //.

# Alla riga 5 troviamo la direttiva #include <iostream>. Questa non è un’istruzione, infatti non termina con il punto e virgola, ma è uno strumento per includere (cioè copiare letteralmente) nel nostro codice il file iostream. I file inclusi si chiamano headers (intestazioni), e permettono al nostro programma di usare le funzionalità messe a disposizione dalle librerie esterne. Il file in questione, iostream, fa parte della libreria standard del C++, e contiene le funzioni necessarie a gestire i flussi di dati d’ingresso e d’uscita (lo streaming). In particolare, la iostream ci permette di utilizzare le funzioni cout (per scrivere sul canale d’uscita primario (la finestra della console), e cin (per ricevere dal canale d’ingresso primario, solitamente la tastiera), come vedremo in seguito.

# Alla riga 7 troviamo l’istruzione using namespace std;. I namespace sono dei contenitori paragonabili alle cartelle usate dai sistemi operativi. I namespace contengono nomi di variabili e funzioni in modo da risolvere eventuali ambiguità. Per accedere alle variabili o funzioni dichiarate in un namespace occorre far uso dell’operatore doppio due punti (::). Per esempio, cin e cout fanno parte del namepsace std e pertanto per poterle utilizzare occorrerebbe scrivere std::cin e std::cout; ma dato che utilizziamo l’istruzione using namespace std; che indica al compilatore di dare per scontato che vogliamo riferirci a variabili appartenenti al namepsace std. Questo rende più semplice la scrittura del codice.

# Alla riga 9 troviamo int main(int argc, char\*\* argv)indicante l’inizio della funzione principale, ovvero il punto d’ingresso dell’applicazione. Un qualsiasi programma deve avere almeno una funzione, la funzione principale, nella quale scrivere il codice che verrà eseguito tutte le volte che viene mandato in esecuzione il programma. Questa riga specifica anche un valore di ritorno di tipo intero (int). Quando un programma termina restituisce al sistema operativo (che l’ha invocato) un valore intero per informarlo se l’esecuzione è andata a buon fine, o meno. Se restituisce 0 significa che il programma è terminato correttamente, altrimenti restituisce un valore diverso da 0. Alla riga 17, infatti, troviamo l’istruzione return(0);. Se il programma, cioè vengono eseguite tutte le istruzioni senza anomalie, arriva alla riga 17, significa che è terminato correttamente e quindi esegue la return(0) appunto per restituire lo 0 al sistema operativo.

# Alle righe 10 e 18 troviamo le parentesi graffe che racchiudono il blocco d’istruzioni della funzione principale main da eseguire.

# Alla riga 15 troviamo l’istruzione cout<<endl; dove endl è un manipolatore esterno appartenente al namepsace std che serve ad inserire una sequenza di fine riga, ovvero manda il cursore a capo.

# Questo è quello che si vede a video mandando in esecuzione il programma:

#

# Esercizio

# Scrivere un programma che stampi su tre linee distinte, le parole “home sweet home”.

# GLI IDENTIFICATORI

Fino ad ora abbiamo costruito programmi che utilizzano il computer come macchina da scrivere; ovviamente il computer può fare molto di più. Per poter apprendere come costruire programmi più complessi dobbiamo vedere prima di tutto come vengono gestiti i dati, in modo da effettuare le operazioni di input e l’elaborazione.

Partiamo, quindi, dagli identificatori. Gli identificatori sono i **nomi** utilizzati per rappresentare variabili, costanti, tipi, e funzioni del programma. Si crea un identificatore specificandolo nella dichiarazione di una variabile, di un tipo o di una funzione. Dopo la dichiarazione, l’identificatore potrà essere utilizzato nelle istruzioni del programma.

Un identificatore è formato da una sequenza di una o più lettere, cifre o caratteri (maiuscoli o minuscoli), senza caratteri di altro tipo o spazi bianchi (a parte l'underscore "\_", che è considerato una lettera) 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.

Non sono validi gli identificatori che coincidono con le parole-chiave del linguaggio (come da Tabella sotto riportata).

Esempi di identificatori validi:

hello

deep\_space9

a123

\_7bello

Esempi di identificatori non validi:

un amico (contiene uno spazio)

un'amica (contiene un apostrofo)

7bello (il primo carattere non è una lettera)

for (è una parola-chiave del C++)

Il linguaggio C++ distingue le lettere maiuscole dalle lettere minuscole; si dice che il C++ è **case sensitive**. Ciò 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

contatore-1

È 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 QWERTY.

Sono entrati a far parte della programmazione comune gli identificatori i,j,k che vengono spesso utilizzati come indici contatori nelle iterazioni.

**Dichiarazione obbligatoria degli identificatori**

In C++ tutti gli identificatori di un programma **devono** essere dichiarati prima di essere utilizzati (non necessariamente all'inizio del programma), cioè deve essere specificato il loro tipo. Per dichiarare un identificatore bisogna scrivere un'istruzione apposita in cui l'identificatore è preceduto dal tipo di appartenenza.

Esempio:

int Variabile\_Intera;

Più identificatori dello stesso tipo possono essere dichiarati nella stessa istruzione e separati l'uno dall'altro da una virgola.

Esempio:

int ore, giorni, mesi;

# LE VARIABILI

Una **variabile** può essere considerata come una scatola che può contenere un solo **oggetto** (testo, numero, immagine, video, etc.). Un oggetto è rappresentato e conservato nella memoria del calcolatore sotto forma di **dato**.



Ogni scatola deve avere un **nome univoco** al fine di poterla distinguere dalle altre.

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.

La “**dichiarazione di una variabile**” è necessaria per 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 (predefiniti) 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.

* Il testo, tipo **char**, può essere formato da serie di caratteri (b, F, !, ?, 6) e stringhe (“Siate affamati, siate folli”). Il tipo char occupa normalmente 8 bit o, in altri termini, 1 byte per carattere. Va detto, anche che tale tipo può essere anche utilizzato per rappresentare valori numerici compresi nell’intervallo tra -128 e 127.
* I valori interi, tipo **int**, sono i valori numerici con i quali si è imparato a contare (1, 2, 43, -89, 4324, ecc.). Normalmente il tipo int occupa 16 bit (o 2 byte) e può quindi contenere valori compresi tra -32768 e 32767. Sui sistemi operativi Windows a 32 bit (Windows 95, Windows 98 e Windows NT) il tipo int occupa invece 32 bit e quindi contiene valori compresi tra -2.147.483.648 e 2.147.483.647.
* I valori in virgola mobile (tipo **float**) sono utilizzati per rappresentare i numeri decimali. Questi numeri sono rappresentati da una parte intera ed una parte frazionale. I numeri in virgola mobile richiedono una maggiore precisione rispetto agli interi e sono quindi normalmente rappresentati da 32 bit. Il loro intervallo varia, perciò, tra 3,4 x 10-38 e 3,4 x 1038 (con 7 cifre significative).
* I valori in virgola mobile a doppia precisione (tipo **double**) sono valori molto estesi che normalmente occupano 64 bit (o 8 byte) e possono avere, quindi, un valore compreso fra 1,7 x 10-308 e 1,7 x 10308 (con 15 cifre significative). I valori long double sono ancora più precisi e normalmente occupano 80 bit (o 10 byte). Il loro valore è compreso fra 1,18 x 10-4932 e 1,18 x 104932 (con ben 19 cifre significative).
* I tipi enumerativi, tipo **enum**, sono definiti dall’utente.
* Il tipo **void** indica valori che occupano 0 byte e non hanno alcun valore.
* Il tipo **puntatore** non contiene vere e proprie informazioni ma l’indirizzo di una cella di memoria contenente tali informazioni. Vedremo meglio tale concetto in seguito.





# I CARATTERI

Qualunque lingua parlata e scritta, per costruire le proprie frasi fa uso di una serie di caratteri. Per esempio, questa guida è scritta utilizzando svariate combinazioni di lettere dell’alfabeto, cifre e segni di punteggiatura. Il tipo char, quindi, serve proprio ad identificare uno degli elementi del linguaggio. In particolare il tipo char potrà contenere:

Una delle 26 lettere minuscole dell’alfabeto:

 **a b c d e f g h i j k l m n o p q r s t u v w y z**

Una delle 26 lettere maiuscole dell’alfabeto:

**A B C D E F G H I J K L M N O P Q R S T U V X Y Z**

Una delle 10 cifre (intese come caratteri e non come valori numerici)

**0 1 2 3 4 5 6 7 8 9**

O, infine, uno dei seguenti simboli:

**+ – \* / = , . \_ : ; ? ” ‘ ~ | ! # % $ & ( ) [ ] { } ^ Â®**

Vediamo adesso un semplice programma C++ che illustra la dichiarazione e l’uso del tipo char:

/\*
\* Un semplice programma C++ che mostra
\* l’uso del tipo char utilizzato sia come contenitore di caratteri \* che come contenitore di valori numerici interi
\*/

#include <iostream>

#include <cstdlib>

using namespace std;

int main(int argc, char\*\* argv){

 char carattere;

 char num;

 cout<<"Inserire un carattere a scelta e premere INVIO: ";

 cin>>carattere;

 cout<<"Inserire un intero compreso tra 0 e 127: ";

 cin>>num;

 cout<<"Il carattere inserito e': "<<carattere<<endl;

 cout<<"Il numero inserito e': "<<num;

 return 0;

}

# APPROFONDIMENTO - I/O IN C - PRINTF - SCANF

/\*
\* Un semplice programma C++ che mostra
\* l’uso del tipo char utilizzato sia come contenitore di caratteri \* che come contenitore di valori numerici interi
\*/

#include <stdio.h>
main(){

char carattere;
char num;
printf(“Inserire un carattere a scelta e premere INVIO: “);
scanf(“%c”,& carattere);
printf(“Inserire un intero compreso tra 0 e 127: “);
scanf(“%d”,& num);
printf(“Il carattere inserito e’: %cn”,carattere);
printf(“Il numero inserito e’: %dn”,num);
return(0);

}

L’esempio appena visto utilizza le funzioni **printf e scanf** che sono derivate dal linguaggio C. La funzione printf viene qui utilizzata per visualizzare sullo schermo le informazioni desiderate, mentre la scanf è usata per leggere l’input dell’utente.

Entrambe queste funzioni sono definite nel file d’intestazione **stdio.h**, ed entrambe utilizzano dei codici di formattazione a seconda del tipo di variabile che l’utente tratta.
Il codice **%c** indica che si sta trattando un carattere mentre il codice **%d** è utilizzato con gli interi.

Quando si scrive: scanf(“%c”,&carattere) si intende dire che si vuole leggere l’input dell’utente come carattere (quindi verrà considerato soltanto il primo elemento immesso) e memorizzare tale valore nella variabile carattere.

Invece, la riga scanf(“%d”,&num) viene utilizzata per leggere un intero decimale e memorizzarlo nella variabile num.

Similmente, quando si scrive:

printf(“Il carattere inserito e’: %cn”,carattere)

si sta stampando sullo schermo la stringa: “Il carattere inserito è: ” più il char identificato dalla variabile carattere e dal codice di formattazione %c.. Ovvero, se si è inserito il carattere ‘c’ verrà, quindi, stampato:

Il carattere inserito è c.

Stesso discorso per l’ultima istruzione del programma:

printf(“Il numero inserito e’: %dn”,num);

dove invece viene stampato un intero identificato dal codice %d e dalla variabile num.

Quello che va notato, è che sia la variabile carattere che la variabile num sono entrambi char. Nel primo caso, come visto, la variabile char carattere viene utilizzata per conservare un carattere. Nel secondo caso, la variabile char num viene utilizzata per conservare un intero, che però non può essere più grande di 127 (si legga la spiegazione di ciò nel paragrafo successivo).

Infine, va detto che il codice di formattazione nel caso di variabili float o double è %f.

# I TIPI DERIVATI

* Il tipo **string** non è uno standard e può essere utilizzato per gestire dati costituiti da sequenze alfabetiche e alfanumeriche.

# GLI INTERI

Il C++ presenta una buona varietà di tipi numerici interi, poiché ai tipi fondamentali:

* **char** intero che occupa un byte (8 bit)
* **int** intero che occupa due byte (16 bit)

permette di aggiungere tre qualificatori unsigned, short, long, (che significano rispettivamente senza segno ,corto e lungo) che ne ampliano il significato. Sappiamo che un char è rappresentato da un byte; un byte è, a sua volta, rappresentato da 8 bit. Ed il bit (che rappresenta l’acronimo di binary digit, ovvero cifra binaria) può contenere i valori 0 oppure 1. Vediamo alcuni esempi di Byte:

10100010
00001100
00000000
11111111

Si definisce bit più significativo quello all’estrema sinistra mentre il bit meno significativo sarà quello all’estrema destra. Nella notazione normale, cioè senza l’uso di qualificatori, il bit più significativo viene utilizzato per il segno; in particolare se il primo bit vale 1 rappresenterà il segno meno mentre se il primo bit vale 0 rappresenta il segno +.

Quindi, invece di 8 bit per rappresentare un valore, ne verranno usati solo 7. E con 7 bit, il valore maggiore che si riesce a raggiungere è 128. Poiché anche lo zero deve essere considerato tra i valori possibili, allora otterremo proprio l’intervallo da 0 a 127 per i positivi e da -1 a -128 per i negativi.

Se viene, invece, utilizzato il qualificatore **unsigned**, tutti i bit che compongono il byte vengono utilizzati per contenere e comporre un valore, che in tal caso potrà essere soltanto un valore positivo. Il numero relativo all’ultimo byte rappresentato nell’esempio precedente varrà, in tal caso:

**1 x 20 + 1 x 21 + 1 x 22 + 1 x 23 + 1 x 24 + 1 x 25 + 1 x 26 + 1 x 27 = 255**

Questo ragionamento è valido per tutti gli altri tipi unsigned del C++.

Il qualificatore short, equivale fondamentalmente al tipo stesso. Dire int o dire short int è, in genere equivalente. Il qualificatore long è invece usato per incrementare il range di valori che solitamente un int può contenere (nel caso dei sistemi operativi Windows a 32 bit i tipi int e long int sono identici). Con tale qualificatore, vengono aggiunti due byte e si passa, dunque, dai 16 bit standard ai 32 bit.

Si noti come unsigned int, short int e long int possano essere dichiarati anche più sinteticamente come unsigned, short, long in quanto mancando l’operando, il qualificatore assume che sia sempre int.

# NUMERI IN VIRGOLA MOBILE

Il C++ consente l’uso di tre tipi di numeri in virgola mobile[[3]](#footnote-3): float, double e long double. Anche se lo standard ANSI C++ non definisce in modo preciso i valori e la quantità di memoria da allocare per ognuno di questi tipi, richiede però che ognuno di essi contenga come minimo i valori compresi fra 1E -37 e 1E+37. La maggior parte dei compilatori C era già dotata dei tipi float e double. Il comitato ANSI C ha poi aggiunto un terzo tipo chiamato long double. Vediamo un esempio che illustra la dichiarazione e l’uso delle variabili float.

/\*
\* Un semplice programma che mostra l’uso del tipo di
\* dati float calcolando l’area di un cerchio
\*/

#include <iostream>

using namespace std;
int main(int argc, char\*\* argv){

float raggio, float pigreca = 3.14, float area;
cout << "Inserire il raggio: ";
cin >> raggio;
cout << endl;
area = raggio \* raggio \* pigreca;
cout << "L'area del cerchio e': " << area << endl;

system("PAUSE");

return(0);

}

# DICHIARAZIONE DI VARIABILI

Come già detto, il C++ richiede tassativamente che ogni variabile prima di essere utilizzata dal programma venga preventivamente dichiarata. La dichiarazione avviene semplicemente indicando il tipo dell’identificatore in oggetto seguito da uno o più identificatori di variabile, separati da una virgola e seguiti dal punto e virgola al termine della dichiarazione stessa.

Per esempio per definire di tipo float le variabili a e b basta indicare:

float a, b;

Oppure per indicare di tipo int le variabili c e d:

int c, d;

E’ importante sottolineare il fatto che una variabile può essere dichiarata soltanto una volta e di un solo tipo all’interno dell’intervallo di azione della variabile stessa.

E’ possibile inizializzare una variabile, cioè assegnare alla stessa un valore contemporaneamente alla sua dichiarazione; ad esempio è consentita la dichiarazione:

float a, b = 4.6; **//il punto anglosassone è la virgola italiana**
char ch = ‘r’; **//singoli apici**

string nome = "luca"; **//doppi apici**

che corrisponde a scrivere:

float a, b;
char ch;
b = 4.6;
ch = ‘r’;

string nome;

nome = "luca";

cioè ad a non verrebbe per il momento assegnato alcun valore, mentre b avrebbe inizialmente il valore 4.6, ch avrebbe il carattere ‘r’, e nome avrebbe la stringa “luca”.

**NB**

Una variabile non può essere definita più di una volta in un contesto.

Una variabile deve essere definita prima di utilizzarla.

Una variabile non necessariamente deve essere definita all’inizio del programma.

**Esempio** - Qual è l’output di questo programma?

#include <iostream>

#include<stdlib.h>

using namespace std;

int main(int argc, char\*\* argv){

 char ch1='C', ch2='I', ch3='A', ch4='O';

 cout<<ch1<<ch2<<ch3<<ch4<<endl;

 system("PAUSE");

 return(0);

}

# BLOCCHI DI ISTRUZIONE

Le **parentesi graffe** { e } contengono blocchi di istruzioni. I blocchi di istruzioni possono consistere nella definizione del corpo di una funzione (come accade nel caso della funzione main() ) oppure nel riunire più linee di codice che dipendono dalla stessa istruzione di controllo, come nel caso in cui diverse istruzioni vengono eseguite a seconda della validità o meno del test di un’istruzione.

# LE COSTANTI

In molti casi è utile assegnare a degli identificatori dei valori che restino costanti durante tutto il programma e che non possano essere cambiati nemmeno per errore. In C++ è possibile ottenere ciò in due modi con due risultati leggermente diversi:

* Con la direttiva al compilatore #define
* Con il modificatore const

# LA DIRETTIVA #define

Sintassi di #define

#define <identificatore> <valore>

Con la direttiva #define il compilatore in ogni punto dove incontra i simboli così definiti sostituisce ai simboli stessi i valori corrispondenti.

**Esempio**

#include <iostream>
#define MAXNUM 10
#define MINNUM 2
using namespace std;

int main(int argc, char\*\* argv){

…
int x,y;
x = MAXNUM;
y = MINNUM;

return(0);

}

automaticamente il compilatore definisce:

x = 10;
y = 2;

Volendo, in seguito modificare i valori in tutto il programma basterà modificare una volta per tutte i soli valori indicati con #define.

Si noti che #define non richiede che venga messo il punto e virgola finale e neppure l’operatore di assegnamento (=).

L’utilizzo della direttiva #define non soltanto diminuisce l’occupazione di memoria (non vi è infatti necessità di spazio per le costanti MAXNUM e MINNUM), ma anche rende più veloce il programma che durante l’esecuzione, quando le deve utilizzare, non deve ricercarne i valori.

C’è da dire che #define ha un **utilizzo più generale** che va oltre la definizione di costanti. Essa permette anche la definizione delle cosiddette macro. Vediamo un esempio per capire bene in cosa consistono le macro. Si supponga che in un programma si debbano invertire molte volte i contenuti di alcuni identificatori. Piuttosto che ripetere ogni volta la stessa sequenza di operazioni, viene utile costruirsi un’istruzione apposita come dal seguente programma:

#include <iostream>
#define inverti (x,y,temp) (temp)=(x); (x)=(y); (y)=(temp);

using namespace std;
int main(int argc, char\*\* argv){

float a= 3.0, b = 5.2, z;
int i = 4, j = 2, k;
inverti(a, b, z); // adesso a = 5.2 e b = 3.0
inverti(i,j,k); // adesso i = 2 e i = 4

return(0);
}

# IL MODIFICATORE const

Sintassi di const

const <identificatore> = <valore>;

Si faccia attenzione al fatto che qui viene usato sia l’operatore di assegnamento (=) che il punto e virgola finale. Se il tipo non viene indicato il compilatore assume per default che sia intero (int).

Ecco un esempio di utilizzo del modificatore const:

const int MAXNUM = 10, MINNUM = 2; const float PIGRECO = 3.1415926;

Si noti che, contrariamente a #define, con un solo utilizzo di const è possibile dichiarare più identificatori separandoli con la virgola. Inoltre, il compilatore assegna un valore che non può essere modificato agli identificatori utilizzati (MAXNUM e MINNUM nell’esempio precedente), valore che però è archiviato in una zona di memoria e non sostituito in fase di compilazione, come accade per #define.

# OPERATORI BOOLEANI

Il C++ mette a disposizione un numero superiore di operatori rispetto ad altri linguaggi, ma alcuni di essi risultano non facilmente interpretabili perché i loro simboli non hanno un immediato riferimento mnemonico alla funzione svolta. Vediamo i più importanti operatori del C++.

# Valori booleani: true e false

Prima di parlare degli operatori booleani, è importante sapere cosa è un valore “booleano”. Un booleano può assumere solo due valori: true o false. Nessun altro valore è permesso.

Gli identificatori booleani e le operazioni su di essi sono molto usati. Spesso, in un programma, si rende necessario sapere se una certa condizione è vera (true) oppure falsa (false). Nel primo caso, il corso del programma prenderà una determinata direzione che, invece, non sarebbe intrapresa altrimenti.

Un esempio grafico particolarmente attinente alle operazioni con boolean è rappresentato dalle check box. Se una check box è selezionata si vorrà intraprendere una determinata azione. In caso contrario non si vorrà fare nulla.

La maggior parte dei linguaggi di programmazione contemplano il tipo booleano. La maggior parte dei compilatori C++ riconosce il tipo boolean con la parola chiave bool. Altri, invece, accettano la scrittura boolean. Diamo per buono che il compilatore riconosca il tipo bool. In tal caso, una dichiarazione di una variabile booleana sarà la seguente:

bool flag;

I principali operatori di tipo booleano, ovvero gli operatori che consentono di eseguire operazioni su elementi di tipo bool sono 3:

|  |  |  |
| --- | --- | --- |
| Funzione | Simbolo | Significato |
| AND | && | Congiunzione logica |
| OR | || | Disgiunzione logica |
| NOT | ! | Negazione logica |

Ciascun operatore prende in input uno o due booleani e restituisce in output un altro booleano.

* **AND**, prende in input due operandi e produce in output un booleano, attenendosi al seguente comportamento: Se entrambi gli operatori sono true allora l’output è true; in tutti gli altri casi l’output è uguale a false.
* **OR**, prende in input due operandi e produce in output un booleano, attenendosi al seguente comportamento: Se almeno uno degli operandi è uguale a true, l’output è true; altrimenti, se nessuno dei due operandi è uguale a true l’output sarà false.
* **NOT**, prende in input un solo operando e produce in output un booleano, attenendosi al seguente comportamento: Se l’operando di input è true allora l’output sarà false. Se, invece l’operando di input è false, allora l’output sarà uguale a true. In altri termini, l’operatore di NOT prende un input e ne restituisce l’esatto contrario.

Vediamo alcuni esempi sull’utilizzo dei tre operatori definiti.

// Supponiamo che Gianni sia stanco
bool gianniStanco = true;
// Supponiamo pure che Gianni non sia costretto ad alzarsi presto
bool gianniDeveAlzarsiPresto = false;
// Andrà a letto ora Gianni?
Bool gianniSiCoricaOra = gianniStanco && gianniDeveAlzarsiPresto;
// Il risultato è false

L’esempio precedente è abbastanza semplice da comprendere. La prima variabile bool (gianniStanco) viene inizializzata a true, il che equivale a dire che Gianni è stanco. La seconda variabile bool (gianniDeveAlzarsiPresto) è inizializzata invece a false, il che vuol dire che Gianni non ha la necessità di alzarsi presto la mattina. La terza variabile booleana (gianniSiCoricaOra) è un risultato dell’operazione di AND tra le due precedenti. Ovvero: Gianni deve andare a letto adesso soltanto se è stanco e deve alzarsi la mattina presto. Quindi, l’operatore AND è il più adatto in tale circostanza.

Se, invece, Gianni decidesse di andare a letto se anche soltanto una delle due precondizioni fosse vera allora l’operatore da utilizzare sarebbe l’operatore OR. Avremo in tal caso:

bool gianniSiCoricaOra = gianniStanco || gianniDeveAlzarsiPresto;
// Il risultato è true

Se, ancora, si verifica la condizione:

gianniStanco = true
bool gianniInForma = !gianniStanco
// gianniInForma sarà uguale a false

Ovvero la variabile booleana gianniInForma sarà vera se non è vera quella che identifica Gianni stanco. Questo è un banale esempio dell’operatore NOT.

L’utilizzo degli operatori booleani è perfettamente lecito anche su variabili che non siano bool. **In C++ il valore “0″ equivale a false** e qualunque valore diverso da zero equivale a true. Ad esempio:

int ore = 4;
int minuti = 21;
int secondi = 0;
bool timeIstrue = ore && minuti && secondi;

Poiché il risultato deriva dall’esame dei tre operandi e c’è un valore (secondi) che è uguale a zero (ovvero equivale a false) il risultato dell’espressione è false.

# OPERATORI ARITMETICI

Il linguaggio C++ è dotato di tutti i comuni operatori aritmetici di somma (+), sottrazione (-), moltiplicazione (\*), divisione (/) e modulo (%). I primi quattro operatori non richiedono alcuna spiegazione data la loro familiarità nell’uso comune.

L’operatore modulo, invece, è semplicemente un modo per restituire il resto di una divisione intera. Ad esempio:

int a = 3, b = 8, c = 0, d;
d = b %a; // restituisce 2
d = a % b; // restituisce 3
d = b % c;

// restituisce un messaggio di errore (divisione per zero)

# NB

# Dividendo due numeri interi si ottiene un intero (si ottiene cioè la parte intera):

int x = 9, y = 4, z = x/y ; //**visualizza 2 e non 2.25**

# invece

float x = 9.0, y = 4.0, z = x/y; cout << z; **//visualizza 2.25**

# OPERATORE DI ASSEGNAMENTO

L’operatore di assegnamento in C++, altro non fa che assegnare ad una variabile un determinato valore. È importante dire che un’espressione contenente un operatore di assegnamento può quindi essere utilizzata anche all’interno di altre espressioni, come ad esempio:

x = 5 \* (y = 3);

In questo esempio, alla variabile y viene assegnato il valore 3. Tale valore verrà moltiplicato per 5 e quindi, in definitiva, alla variabile x sarà assegnato il numero 15.

È, però, caldamente sconsigliato l’utilizzo di tale pratica in quanto potrebbe rendere poco leggibili le espressioni. Vi sono, tuttavia, un paio di casi in cui questa possibilità viene normalmente sfruttata. Innanzitutto, si può assegnare a più variabili lo stesso valore, come in:

x = y = z = 4;

Il secondo tipo di utilizzo è molto frequente all’interno dei cicli while e for:

while ((c = getchar()) != EOF){
  …
}

oppure

for(int i = 0; i < 10; i++){
  …
}

# OPERATORI DI RELAZIONE

Un operatore di relazione è un operatore che **verifica una condizione** come: “è minore di” oppure “è maggiore di” oppure ancora “è uguale a”.

Un utilizzo intuitivo di tali operatori è quello che viene fatto per comparare due numeri. Un operatore di relazione può essere determinante per la scelta di una determinata strada piuttosto che di un’altra. Ad esempio, se una certa variabile supera un valore di soglia potrebbe essere necessario chiamare una funzione per controllare altri parametri, e così via.

Ma vediamo nella seguente tabella, l’elenco completo degli operatori di relazione:

|  |  |  |  |
| --- | --- | --- | --- |
| Nome | Simbolo | Esempio | Risultato |
| Minore | < | boolris = (3 < 8) | true |
| Maggiore | > | boolris = (3 > 8) | false |
| Uguale | == | boolris = (5 == 5) | true |
| MinoreUguale | <= | boolris = (3 <= 5) | true |
| MaggioreUguale | >= | boolris = (5 >= 9) | false |
| Diverso | != | boolris = (4 != 9) | true |

# PRECEDENZA TRA OPERATORI

La **precedenza tra operatori** indica l’ordine con cui gli operatori vengono valutati dal **compilatore**. Un operatore con precedenza maggiore verrà valutato per prima rispetto ad un operatore con precedenza minore, anche se quest’ultimo figura prima dell’operatore con precedenza maggiore. Ecco un esempio:

int risultato = 4 + 5 \* 7 + 3;

Il risultato in questo caso dipende proprio dalla precedenza tra operatori. In C++, l’operatore di **moltiplicazione** (\*) ha precedenza rispetto all’operatore **addizione** (+). Quindi la moltiplicazione 5\*7 avverrà prima di tutte le altre addizioni. Ecco la sequenza della risoluzione dell’espressione precedente:

int risultato = 4 + 5 \* 7 + 3;

risultato = 4 + 35 + 3 = 42

Perché le operazioni siano effettuate con ordine diverso, sarà sufficiente introdurre delle **parentesi tonde**. Ad esempio se vogliamo moltiplicare la somma 4+5 con la somma di 7+3, basterà scrivere:

int risultato = (4 + 5) \* (7 + 3);

La variabile risultato, in questo caso, varrà 90.

Facciamo ora un elenco di tutti gli operatori ordinati per livello di precedenza. Ad ogni riga della tabella è assegnato un livello di precedenza. L’operatore a maggior priorità avrà il valore 1 e a seguire quelli a priorità sempre più bassa.

|  |  |  |
| --- | --- | --- |
| Livello di precedenza | Operatore | Nome |
| 1 | ! | Not, negazione |
| 2 | \* | Moltiplicazione |
| 2 | / | Divisione |
| 2 | % | Modulo |
| 3 | + | Addizione |
| 3 | - | Sottrazione |
| 4 | < | Minore |
| 4 | <= | Minore uguale |
| 4 | > | Maggiore |
| 4 | >= | Maggiore uguale |
| 5 | == | Uguale (confronto) |
| 5 | != | Diverso |
| 6 | && | AND |
| 7 | || | OR |
| 8 | = | Assegnamento |

Le **parentesi** risolvono praticamente il problema di conoscere la precedenza tra gli operatori. Per evitare errori ed aumentare la leggibilità del codice può essere consigliabile l’uso delle parentesi in ogni situazione in cui vi sia la presenza contemporanea di più operatori e si abbiano dubbi sulle precedenze.

# L’OPERAZIONE DI ESTRAZIONE TRAMITE CIN

# Abbiamo visto come le inizializzazioni e gli assegnamenti permettono l’associazione di un valore ad una variabile. Tale valore può essere una costante, il contenuto di un'altra variabile o in generale il risultato di un’espressione. In ogni caso, però, l’operazione è frutto di quanto stabilito nel codice, e l’utente finale non ha possibilità di interazione con il programma.

# Spesso, vi è, infatti, la necessità di richiedere all’utente l’inserimento dei dati i cui valori devono essere salvati nelle variabili.

# Per richiedere un valore possiamo utilizzare lo stream cin, che rappresenta il flusso dati in ingresso, solitamente associato alla tastiera.

# Finora abbiamo usato il flusso cout, servendoci dell’operatore di inserimento <<, che permette di agganciare uno stream alla fine di un altro. L’operazione inversa si chiama estrazione (>>) che permette l’estrazione dei dati da uno stream (la tastiera di default). L’estrazione cerca di convertire il dato estratto, coerentemente con il tipo di variabile in cui tale dato verrà inserito.

# La sintassi dell’estrazione è la seguente:

# cin >> variabile;

# Vediamo un esempio tipico d’utilizzo dell’estrattore:

# #include <iostream>

# #include<stdlib.h>

# using namespace std;

# int main(int argc, char\*\* argv){

#  int num, doppio;

#  cout<<"Scrivi un numero intero => ";

#  cin>>num;

#  doppio=num\*2;

#  cout<<"Il numero inserito e' =>"<<num<<endl;

#  cout<<"Il cui doppio e' =>"<<doppio<<endl;

#  return(0);

**}**

# e questo è il risultato:

#

# Esercizio

# Realizzare un programma che simuli le operazioni aritmetiche (addizione, sottrazione, divisione e moltiplicazione) di una classica calcolatrice. Chiedete all’utente di inserire due numeri da tastiera e a questi applicate una delle operazioni indicate.

# L’OPERAZIONE D’INCREMENTO

# Molto utili all’interno dei nostri programmi, sono le variabili di accumulo che permettono l’auto-incremento, ovvero l’incremento di una unità. Vediamo come fare:

# nomeVariabile = nomeVariabile + 1 ;

# Aalla variabile nomeVariabile viene assegnato il contenuto della variabile stessa preventivamente incrementata di uno.

# Questa istruzione può essere scritta anche in quest’altro modo:

# nomeVariabile++;

# Bisogna fare attenzione che se si utilizza l’operatore ++ in una espressione di assegnamento, esiste una sostanziale differenza fra l’uso della notazione prefissa (++nomeVariabile) e quella della notazione postfissa (nomeVariabile++), sebbene scrivere nomeVariabile++ oppure ++nomeVariabile è equivalente a nomeVariabile = nomeVariabile + 1.

# Per capire questa differenza vediamo il seguente esempio:

#  int num1, num2;

#  num1=0;

#  cout<<"num1\tnum2\n";

#  num2=num1++;

#  cout<<num1<<"\t"<<num2<<endl;

#  num1=0;

#  num2=++num1;

#  cout<<num1<<"\t"<<num2<<endl;

# Questo è il risultato:

#

# La differenza nel risultato visualizzato, fra la prima e la seconda riga, è dovuta al fatto che con l’operatore ++ prefisso restituisce prima il valore della variabile, e poi la incrementa. L’operatore postfisso, invece, fa l’inverso, ovvero prima incrementa e poi restituisce.

# In modo analogo esiste anche la possibilità di decrementare la variabile con l’operatore – (prefisso o postfisso).

# Per incrementi maggiori di un’unità e per altre operazioni autoreferenziali (aritmetiche o bot-a-bit), si usano degli simboli composti dall’operatore di partenza seguito dall’uguale.

# Esercizio – verificare il valore assunto da x dopo queste operazioni:

#  int x=5;

#  x+=3; //a=a+3;

#  x%=3; //x=x%3;

#  x^=10; //x=x^10;

# CASTING

# Le operazioni di casting consentono la conversione di un valore da un tipo ad un altro.

# Esempio

#  char ch='a';

#  cout<<(int)ch<<endl;

#  ch++;

#  cout<<(int)ch<<endl;

# In questo esempio il casting viene utilizzato per ottenere la codifica ASCII di un carattere. Si scopre così che alla lettera a corrisponde il codice 97, e alla lettera b il codice 98. Per ottenere la lettera b, a partire dalla lettera a, basta incrementare di uno la variabile del tipo carattere.

# Esercizio

# Nel videogioco PACMAN qual è la condizione da porre per verificare se il personaggio puntiforme si trova alle coordinate x=20 e y=30?

# CONTROLLO DEL FLUSSO

# Il linguaggio di programmazione C++, come altri, prevede l’utilizzo di costrutti che consentono l’alterazione del flusso di esecuzione, facendolo tornare indietro, saltare in avanti (in rari casi), e ramificare.

# Per illustrare ciò graficamente, possiamo far uso dei diagrammi di flusso.

# LA SEQUENZA

# Il diagramma qui riportato rappresenta il flusso lineare, quello che abbiamo usato in tutti gli esempi fino ad ora.

#

# LA SELEZIONE

Il costrutto di selezione ci consente di far eseguire in modo selettivo una singola riga di codice o una serie di righe di codice (che viene detto **blocco di istruzioni**).

Il C++ fornisce quattro istruzioni condizionali:

* **if**
* **if … else**
* **?**
* **switch**

Prima di affrontare tali istruzioni una per una, è bene dire che, quando all’istruzione condizionale è associata una sola riga di codice, non è necessario racchiudere l’istruzione da eseguire fra **parentesi graffe**. Se invece all’istruzione condizionale è associata una serie di righe di codice, il blocco di codice eseguibile dovrà essere racchiuso fra parentesi graffe.

Per evitare il rischio di errori (specie quando si è neofiti) e per una buona lettura del codice è consigliabile l’uso delle parentesi graffe anche quando l’istruzione da eseguire è soltanto una.

# IL COSTRUTTO IF … ELSE

Il costrutto in questione è molto intuitivo se si considera che le parole inglesi **if** ed **else** corrispondono rispettivamente alle italiane se e altrimenti. La sintassi di if è:

if (<condizione>){

 (<istruzioni da svolgere se la condizione è vera>);

}



Mentre se if è accompagnata da altre istruzioni alternative la sintassi è:

if(<condizione>){

 (<istruzioni da svolgere se la condizione è vera>);

}

else{

 (<istruzioni da svolgere se la condizione è falsa>);

}



Oppure se è necessario condizionare anche le istruzioni rette da else:

if(<condizione 1>){

 <istruzioni da svolgere solo se la condizione 1 è vera>);

}

else if(<condizione 2>){

 (<istruzioni da svolgere solo se la condizione 1 è falsa e la condizione 2 è vera>);

}

else{

 (<istruzioni da svolgere solo se la condizione 1 è falsa e la condizione 2 è falsa>);

}



All’interno delle parentesi le istruzioni vanno completate sempre con il punto e virgola, mentre se non utilizziamo blocchi, ma semplici istruzioni facciamo attenzione a non dimenticare il punto e virgola prima dell’istruzione else.

Ad esempio:

if(i>=0)

 <istruzione 1>; // singola istruzione

else{ // blocco di istruzioni

 <istruzione 2>;

 <istruzione 3>;

}

L’istruzione 1 verrà eseguita solo se i >= 0 mentre in caso contrario viene eseguito il blocco con le istruzioni 2 e 3.

# Esempio

# Utilizziamo il costrutto di selezione per alterare il flusso di esecuzione al fine di prevenire errori di digitazione dell’utente, nel nostro esempio occorre calcolare la divisione fra due numeri interi a e b. se l’utente è poco intelligente e assegna un valore nullo al denominatore occorre alterare il flusso visualizzando un messaggio d’errore.

# Ecco l’esempio:

#  int a, b;

#  cout<<"Numeratore: ";

#  cin>>a;

#  cout<<"Denominatore: ";

#  cin>>b;

#  if(b==0)

#  cout<<"Errore, il denominatore e' nullo\n";

#  else

#  cout<<a/b<<endl;

# ISTRUZIONI IF … ELSE NIDIFICATE

Quando si utilizzano più **costrutti if-else** nidificati, occorre sempre essere sicuri dell’azione **else** che verrà associata ad un determinato **if**. Vediamo cosa accade nell’esempio seguente:

if(temperatura < 20)

if(temperatura < 10) cout << "Metti il cappotto!n";

else cout << " Basta mettere una felpa";

Come si vede, il codice non è allineato in maniera ben leggibile ed è facile che il programmatore poco esperto possa confonderne il flusso di esecuzione, non capendo bene a quale istruzione if fa riferimento l’istruzione **else** finale.

La regola base, in questi casi, dice che un’istruzione **else** è sempre riferita all’ultima istruzione **if** che compare nel codice. Ma, è bene non confondersi troppo le idee utilizzando tale regola, poiché ci sono casi in cui appaiono tante istruzioni if-else annidate con la conseguenza che la lettura del codice sarebbe davvero difficile per chiunque.

Per rendere il codice leggibile e per confondersi il meno possibile, è buona norma fare uso sempre delle parentesi graffe e dell’**indentazione** del codice.

Indentare il codice significa inserire degli spazi prima della scrittura di una riga in modo da facilitarne la comprensione e la leggibilità. L’esempio precedente, indentato e corretto con l’uso delle parentesi diventa:

if(temperatura < 20){

 if(temperatura < 10){

 cout << "Metti il cappotto!";

 }

 else{

 cout << " Basta mettere una felpa";

 }

}

Oppure, per chi ama le forme abbreviate (ma più insidiose):

if(temperatura < 20)

 if(temperatura < 10) cout << "Metti il cappotto!";

 else cout << "Basta mettere una felpa";

Ora è decisamente più comprensibile. Adesso, infatti, diventa chiaro che l’istruzione else è riferita alla istruzione if(temperatura < 10).

# Vediamo questo esercizio significativo:

# si vuole intercettare un numero intero trasformando in uno 0 se è negativo, in un 1 se è positivo e in un -1 se è nullo. Osserviamo il seguente codice e cerchiamo di individuare l’errore:

#  int a;

#  cout<<"num: ";

#  cin>>a;

#  if(a>0) a=1;

#  if(a<0) a=0;

#  else a=-1;

#  cout<<a<<endl;

# Se si assegna ad a il valore 5 il risultato che si ottiene è -1, perché? Provate a cercare l’errore.

# LE ISTRUZIONI SWITCH

Le istruzioni switch permettono di evitare noiose sequenze di if. Esse sono particolarmente utili quando in un programma si deve dare all’utente la possibilità di scegliere tra più opzioni. La sintassi di tali istruzioni è la seguente:

switch(<espressione intera>){

case (<valore costante 1>):

…( <sequenza di istruzioni>)
break;

case (<valore costante 2>)

…( <sequenza di istruzioni>)
break;
….
….

default:

// è opzionale
…( <sequenza di istruzioni>)

}

Il costrutto precedente esegue le seguenti operazioni:

* Valuta il valore dell’espressione intera passata come parametro all’istruzione switch.
* Rimanda lo svolgimento del programma al blocco in cui il parametro dell’istruzione case ha lo stesso valore di quello dell’istruzione switch.
* Se il blocco individuato termina con un’istruzione break allora il programma esce dallo switch.
* Altrimenti, vengono eseguiti anche i blocchi successivi finché un’istruzione break non viene individuata oppure non si raggiunge l’ultimo blocco dello switch.
* Se nessun blocco corrisponde ad un valore uguale a quello dell’istruzione switch allora viene eseguito il blocco default, se presente.



Vediamo un paio di esempi per comprendere meglio quanto detto:

bool freddo, molto\_freddo, caldo, molto\_caldo;

cin>>temperatura;
switch(temperatura){

case(10):

freddo = true;
molto\_freddo = false;
caldo = false;
molto\_caldo = false
break;

case(2):

freddo = true;
molto\_freddo = true;
caldo = false;
molto\_caldo = false;
break;

case(28):

freddo = false;
molto\_freddo = false;
caldo = true;
molto\_caldo = false;
break;

case(40):

freddo = false,
molto\_freddo = false;
caldo = true;
molto\_caldo = true;
break;

default:

freddo = false;
molto\_freddo = false;
caldo = false;
molto\_caldo = false;

}

Nell’esempio precedente, viene valutato il valore costante temperatura passato come parametro a switch. A seconda del valore che tale costante assume viene poi eseguito il relativo blocco di istruzioni. Se, invece, nessuno dei blocchi ha un valore uguale a quello passato a switch, verrà eseguito il blocco default.

Si noti, che nel caso seguente:

………..
case(28):

freddo = false;
molto\_freddo = false;
caldo = true;
molto\_caldo = false;

case(40):

freddo = false,
molto\_freddo = false;
caldo = true;
molto\_caldo = true;
break;
…………….

se il valore della temperatura fosse di 28 gradi, verrebbe giustamente eseguito il blocco corrispondente, ma poiché manca l’istruzione break alla fine di tale blocco, il programma avrebbe continuato ad eseguire anche le istruzioni del blocco successivo (quello con la temperatura uguale a 40 gradi) e quindi il valore finale delle variabili sarebbe stato:

freddo = false;
molto\_freddo = false;
caldo = true;
molto\_caldo = true;

# LE ITERAZIONE

# Abbiamo tre diversi costrutti per realizzare l’iterazione:

# Iterazione con condizione iniziale: ciclo WHILE.

# Iterazione con condizione finale: ciclo DO … WHILE.

# Iterazione con contatore: ciclo FOR.

# Vediamoli.

# IL CICLO WHILE

Nel costrutto WHILE *condizione* rappresenta un controllo booleano che viene effettuato ogni volta al termine del blocco di istruzioni contenuto tra le parentesi graffe.

Se la condizione restituisce true allora il ciclo sarà eseguito ancora mentre se la condizione ritorna un valore uguale a false il ciclo sarà terminato.

L’istruzione while segue la seguente sintassi:

|  |  |
| --- | --- |
| //Blocco\_A//inizio del ciclowhile(condizione){ // Istruzioni da iterare //Blocco\_B }//fine del ciclo//Blocco\_C |  |

**Esempio 1**

Supponiamo di voler scrivere un programma che stampi sullo schermo tutti i numeri pari tra 2 e 15:

#include <iostream>

using namespace std;

int main(int argc, char\*\* argv){

// Definiamo una variabile che conterrà il valore corrente
int numeroCorrente = 1;
// ciclo while che stampa sullo schermo tutti i numeri pari
// tra 1 e 15
while (numeroCorrente < 15){

//cerr è simile a cout; viene utilizzato per visualizzare

//messaggi di errore

cerr << numeroCorrente << endl;
numeroCorrente = numeroCorrente + 2;

}
cerr << “Fine del Programma!” << endl;

return(0);

}

Il funzionamento del programma precedente è molto semplice: viene definita una variabile numeroCorrente che useremo per conservare il valore che di volta in volta il ciclo WHILE modificherà. Inizialmente tale variabile viene inizializzata a 2 (che è il primo valore pari dell’intervallo). A questo punto si entra nel ciclo WHILE e viene testata la condizione che si chiede se numeroCorrente è minore di 15. Ovviamente la condizione viene soddisfatta e così viene eseguito il blocco all’interno del ciclo WHILE. La prima istruzione del blocco stampa il valore della variabile numeroCorrente (che è attualmente 2) mentre la seconda istruzione incrementa il valore della stessa variabile di 2 unità. Adesso numeroCorrente vale 4.

Si ricomincia di nuovo: si testa la condizione, questa è ancora soddisfatta, si stampa il valore della variabile (adesso è 6) e si incrementa numero\_corrente di 2 unità (ora vale 8). E così via fino a quando la condizione non è più verificata, ovvero quando accadrà che numero corrente varrà 16.

L’output del programma precedente, sarà allora il seguente:

2
4
6
8
10
12

14
Fine del Programma!

**IL CICLO DO-WHILE**

Pur essendo molto simili tra loro i due cicli, DO…WHILE e WHILE, si può osservare una marcata differenza nel numero di volte che le istruzioni, contenute nel ciclo, sono eseguite. Il ciclo DO…WHILE esegue le istruzioni contenute nel ciclo stesso almeno una volta. Infatti, come si vede dalla sintassi, il controllo della condizione viene eseguito al termine di ogni loop. Viceversa il ciclo WHILE potrebbe non eseguire mai le istruzioni del ciclo se la condizione risulta da subito falsa.

Il ciclo DO…WHILE ha la seguente sintassi:

|  |  |
| --- | --- |
| //Blocco\_A//inizio del ciclodo{ // Istruzioni da iterare //Blocco\_B}while(condizione);//fine del ciclo//Blocco\_C |  |

Vediamo nella seguente vignetta la differenza fra questi due tipi di cicli:



Analizzando i due cicli si comprende la sostanziale differenza:

|  |  |
| --- | --- |
| C:\Users\User\Desktop\RedmiNote5\Facebook\FB_IMG_1546416850853.jpg | while(non bordo){esegui();} |
| C:\Users\User\Desktop\RedmiNote5\Facebook\FB_IMG_1546416850853.jpg | do{esegui();}while(non bordo); |

**Esempio 1**

Se volessimo scrivere lo stesso esempio della visualizzazione dei numeri pari compresi tra 1 e 15 otterremmo:

#include <iostream>

using namespace std;

int main(int argc, char\*\* argv)//input/output cout{

// Definiamo una variabile che conterrà il valore corrente
int numeroCorrente = 1;
// ciclo do-while che stampa sullo schermo tutti i numeri pari
// compresi fra 1 e 15
do{

cout << numeroCorrente << endl;
numeroCorrente = numeroCorrente + 2;

}while(numero\_corrente < 16);
cerr << “Fine del Programma!” << endl;

 return(0);

}

Il funzionamento del programma precedente è molto simile a quello visto per il ciclo WHILE con la differenza, come detto, che la condizione viene verificata al termine dell’esecuzione del blocco di istruzioni. Se, per esempio, il valore iniziale della variabile numeroCorrente fosse stato 26, il programma avrebbe comunque stampato sul video il valore 26 e poi si sarebbe fermato. In tal caso, avremmo ottenuto un risultato diverso da quello desiderato in quanto il numero 26 è certamente maggiore del 23 che rappresenta il limite dell’intervallo da noi definito. Il ciclo WHILE, invece, non avrebbe stampato alcun numero.

Tale osservazione va tenuta presente per capire che non sempre le tre istruzioni di ciclo sono intercambiabili.

**Esempio 2**

Calcolare la media di 10 numeri interi scelti dall’utente.

 int num, somma=0, media, conta=0;

 const int MAX=10;

 do{

 cout<<"Inserisci un numero: ";

 cin>>num;

 somma=somma+num;

 conta++; //conta=conta+1

 }while(conta<MAX);

 media=somma/MAX;

**Esempio 3**

Calcolare la media di N numeri interi con N scelto dall’utente.

 int num, media=0, conta=0, N;

 cout<<"Quanti numeri da elaborare: ";

 cin>>N;

 do{

 cout<<"Inserisci un numero: ";

 cin>>num;

 media+=num;

 conta++; //conta=conta+1

 }while(conta<N);

 media/=N;

 cout<<"Media: "<<media<<endl;

# IL CICLO FOR

Il ciclo for ha la seguente sintassi:

|  |  |
| --- | --- |
| //Blocco\_Afor(inizializzazione; condizione; step){(<istruzioni da eseguire all’interno del ciclo>)//Blocco\_B}//fine del ciclo//Blocco\_C |  |

Il FOR è costituito da 3 parametri (separati dal punto e virgola)

* Il primo parametro, inizializzazione, setta il valore iniziale del contatore del ciclo.
* Il secondo parametro, condizione, rappresenta la condizione da testare ad ogni passo del ciclo per vedere se è necessario continuare oppure uscire dal ciclo stesso.
* Il terzo parametro, step, descrive la modifica che viene apportata al contatore del ciclo ad ogni esecuzione.

Il ciclo FOR è sintatticamente identico al ciclo WHILE.

Ecco un esempio:

// Il codice seguente esegue la somma dei primi 10 numeri interi

int totale=0; // Questa variabile contiene la somma totale

int i; //dichiaro la variabile controllo del ciclo

// Il ciclo seguente aggiunge i numeri da 1 a 10

// alla variabile totale
for(i=1; i < 11; i++){

totale = totale + i;

}

Quindi, nella porzione di codice precedente accade che:

il parametro inizializzazione è un intero che vale 0, la condizione verifica che i sia minore di 11 per continuare il ciclo, e lo step aggiunge 1 ad ogni passo del ciclo al inizializzazione (l’istruzione i++ serve per incrementare di un’unità un intero).

NB il nome del linguaggio, C++, deriva proprio dalla notazione di incremento unitario.

Subito dopo l’esecuzione iniziale del primo loop la variabile intera i viene settata a 1, viene eseguita l’istruzione totale = totale + i e quindi il valore della variabile totale diviene uguale ad 1. A questo punto viene eseguito l’incremento e quindi i diventa uguale a 2. Viene eseguita la condizione di test e quindi ancora incrementata la variabile i al valore 2. totale ora varrà allora 1 + 2 = 3.

Si prosegue così finché la variabile i resta minore di 11. In definitiva avremo eseguito l’operazione:

1+2+3+4+5+6+7+8+9+10 = 55.

NB for(parametro1, parametro2, parametro3) non è un’istruzione e quindi non termina con un punto e virgola.

**GLI ARRAY A UNA DIMENSIONE**

Gli array sono tipi di dati strutturati atti a rappresentare vettori e matrici, e vengono utilizzati per memorizzare una serie di elementi fra loro omogenei (nel linguaggio C/C++ tutti dello **stesso tipo**).

**Esempio 1**

Consideriamo ad esempio una via cittadina, e individuiamo tutti le abitazioni in base al loro numero civico:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Numero civico | 1 | 2 | 3 | 4 | 5 |
| Abitazione | Abitazione di Paolo Rossi | Abitazione di Luca Verdi | Abitazione di Anna Neri | Abitazione di Elena Grigi | Abitazione di Rita Gialli |

È chiaro quindi che conoscendo il numero civico di un’abitazione il postino è in grado di consegnare la posta alla abitazione corretta.

Si può quindi dire che la struttura realizzata è un vettore (array a 1 dimensione) avente le seguenti caratteristiche:

* L’array ha un nome: “Abitazione”.
* L’array è costituito da un numero finito di elementi adiacenti (nel caso in esame 5).
* Attraverso il numero civico è possibile individuare un elemento all’interno dell’array.

**Esempio 2**

Si può pensare al vettore come ad un insieme di cassetti numerati: per poter accedere al contenuto di un cassetto deve essere specificato il numero ad esso associato (l’**indice**), con la particolarità però che il primo cassetto ha indice 0 e l’ultimo ha indice N-1 se il numero di elementi è pari a N.

Nel seguente esempio è stato creato un array che si chiama VOTI avente dimensione N = 5:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| I | 0 | 1 | 2 | 3 | 4 |
| VOTI | 6.5 | 7.75 | 5.75 | 4.50 | 7 |

Pertanto, il primo elemento, individuato dall’indice I = 0, ha valore pari a 6.5, il secondo 7.75, e così via, fino all’ultimo elemento individuato dall’indice I = 4, il 7.

**NB L’array** ad una dimensione (conosciuto anche come **vettore**).

**Le proprietà fondamentali di un array informatico**

Possiamo dire che un array ha le seguenti proprietà fondamentali:

* Ad ogni array viene assegnato un **nome** univoco che lo identifica, per esempio VOTI, e abbastanza indicativo del suo campo di utilizzo (Il nome dell’array è un valore costante che rappresenta l’indirizzo di memoria del primo elemento dell’array stesso).
* Gli oggetti che compongono l’array sono denominati **elementi**.
* Il numero di oggetti dell’array deve essere **finito**, per esempio N.
* Tutti gli elementi di un array devono essere dello **stesso tipo**. Per esempio o solo numeri interi (INT), o solo numeri reali (FLOAT), o solo stringhe di caratteri (STRING).
* Tutti gli elementi di un array vengono memorizzati uno di seguito all’altro nella memoria del calcolatore e per tale motivo occupano delle locazioni di memoria chiamate **celle**. Spesso si dice che l’array è costituito da N celle piuttosto che N elementi.
* Attraverso un indice, per esempio I, è possibile accedere ad un elemento dell’array.
* L’indice I deve essere unicamente di **tipo numerico**.
* Il **primo elemento dell’array ha indice I = 0**.
* **L’ultimo elemento dell’array ha indice I = N – 1 (dove N è il numero di elementi)**

**Esempio 3**

Un altro esempio tipico è quello relativo alla gestione dei dati memorizzati nella memoria RAM del calcolatore: in ogni cella della memoria può essere salvato un dato (o una istruzione di un programma in corso di esecuzione) e attraverso un indice è possibile accedere alla cella stessa per estrarne il valore o viceversa per scriverci un nuovo valore (sovrascrivendo il precedente valore).

**Esempio 4**



I

16

20

14

18

15

 0 1 2 3 4

Vediamo alcuni altri esempi di array (multidimensionali):

  

# DICHIARAZIONE DI UN ARRAY

Abbiamo visto che, nelle variabili semplici, per accedere al valore contenuto in esse è necessario specificare il nome e, inoltre, una variabile con un valore diverso avrà un nome diverso.

La sintassi per la dichiarazione di una variabile è la seguente:

tipo nome\_variabile;

 tipo nome\_variabile = valore;

nome\_variabile = valore;

Esempio:

int x;

 4

 int x = 4;

 x

x = 4;

Nel vettore esiste un nome che però stavolta identifica il vettore come struttura (il nome farà riferimento all’insieme dei cassetti dell’esempio precedente: questo concetto sarà chiarito meglio più avanti). I singoli elementi del vettore verranno referenziati specificando la loro posizione relativa all’interno della struttura (l’indice): l’elemento a cui ci si riferisce dipende dal valore assunto dall’indice.

Gli elementi del vettore vengono allocati in posizioni di memoria adiacenti e, per tale motivo, è necessario dichiarare un vettore prima del suo uso in modo che il compilatore possa riservare in memoria lo spazio necessario per conservarlo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| **4** | **6** | **3** | **2** | **1** | **7** |

La sintassi per la dichiarazione di un vettore (array) a una dimensione è:

tipo nome\_variabile[dimensione];

dove tipo è il tipo base del vettore, ovvero il tipo di ogni elemento del vettore stesso; e il numero di elementi del vettore è definito da dimensione.

Il seguente esempio dichiara un vettore chiamato numero contenente un massimo di 10 elementi interi:

int numero[10];

Altri esempi:

int array\_uno[10]; // Un array di 10 interi
char array\_due[20]; // Un array di 20 caratteri

Si noti che all’interno delle parentesi quadre non è possibile, in fase di dichiarazione dell’array, utilizzare nomi di variabili.

Poiché il C assegna gli indici agli elementi di un array a partire da 0, gli elementi del vettore di interi appena dichiarato vanno da numero[0] a numero[9].

Quindi, per quanto osservato precedentemente: numero è il nome della struttura, identifica l’insieme degli elementi cioè 10 numeri interi, numero[0], numero[5] individuano il primo ed il sesto numero della struttura.

Se si utilizza numero[i], l’elemento a cui ci si riferisce dipende dal valore assunto dalla variabile i che deve essere di tipo int.

La dimensione di un vettore deve essere necessariamente, e ovviamente, un numero intero, pertanto una notazione come la seguente è errata:

int numero[10,5]; //errata

perché non esiste la cella di posizione 10,5.

Altro esempio:

char lettere[4];

Si definisce un vettore, il cui nome è **lettere**, di 4 elementi del tipo carattere.

Altri esempi:

char nome[]; // Invalido

char nome[0]; // Invalido

char nome[-1]; // Invalido

char nome[10]; // Valido

Ancora:

float numReali[20] ;

Si definisce un vettore, il cui nome è **numReali**, di 20 elementi del tipo float (numeri reali a singola precisione).

Questi tipi di array vengono detti ad **una dimensione** perché per individuare un elemento all'interno della struttura, basta specificare un solo indice.

Si noti che all’interno delle parentesi quadre non è possibile, in fase di dichiarazione dell’array, utilizzare nomi di variabili. Per esempio la seguente dichiarazione non è ammessa:

int DIM ;

char vet[DIM] ;

È possibile, per specificare le dimensioni di un array, usare delle costanti definite, come ad esempio:

#define DIM 10 //direttiva che consente di definire una costante DIM avente valore 10

int dati[DIM]; //la dimensione del vettore è pari alla costante DIM

Volendo, in seguito modificare i valori in tutto il programma basterà modificare una volta per tutte i soli valori indicati con #define.

Si noti che #define non richiede che venga messo il punto e virgola finale e neppure l’operatore di assegnamento (=).

L’utilizzo della direttiva #define non soltanto diminuisce l’occupazione di memoria (non vi è infatti necessità di spazio per la costante DIM), ma anche rende più veloce il programma che durante l’esecuzione, quando le deve utilizzare, non deve ricercarne i valori.

Altro modo:

const int DIM=10 ; //modificatore const che permette la definizione di una costante DIM

//avente valore 10

double num[DIM] ; //la dimensione del vettore è pari alla costante DIM

Il linguaggio C non esegue una verifica sui limiti dell’array, pertanto esiste la possibilità di superarli. Se ciò si verifica, per esempio in un assegnamento, si corre il rischio di alterare un'altra variabile o un pezzo di codice del programma. In altri termini si può utilizzare un array di dimensione n oltre i suoi limiti senza che vengano segnalati errori di compilazione o di esecuzione, ma con il risultato che probabilmente sarà il programma a mostrare dei malfunzionamenti.

Il C++ (e lo stesso C), infatti, non effettua nessun controllo sugli indici che si utilizzano quando si eseguono operazioni sugli array. Per cui, ad esempio, se eseguiamo il seguente codice:

int vet[10];

……
…… // L’array viene inizializzato

int i;

for(i=0; i < 14; i++){

cout << vet[i] << endl;

}

il risultato sarà corretto fino a quando la variabile i sarà minore o uguale a 9 (infatti la dimensione dell’array è stata definita essere uguale a 10). Quello che avverrà, quando il programma cercherà di andare a leggere i valori vet[10], vet[11], vet[12] e vet[13] è assolutamente casuale. Infatti, il programma andrà a leggere delle aree di memoria che non sono state inizializzate o, addirittura, che contengono dei valori relativi ad altre variabili (anche di altri programmi caricati nella RAM), stampando quindi sullo schermo risultati inaspettati.

Se si fosse usata una costante per definire la dimensione dell’array questo errore sarebbe stato facilmente evitato. Infatti:

#define DIM 10

int vet[DIM];

……
…… // L’array viene inizializzato

int i;

for(i=0; i < DIM; i++){

cout << vet[i] << endl;

}

Come si può notare facilmente in questo modo il programmatore non deve preoccuparsi di ricordarsi la dimensione dell’array vet poiché la costante DIM viene utilizzata proprio per tale scopo, evitando i cosiddetti errori di “**array out of bounds**”.

**INIZIALIZZAZIONE DI UN VETTORE**

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 numeri[3] = {12, 0, 4};

char vocali[5] = {'a', 'e','i','o','u'};

float decimali[2] = {1.329, 3.34};

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 poiché 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.

Vediamo un altro esempio che ci permette di capire l’utilità dell’utilizzo dei vettori:

si vuole calcolare la media su 10 interi si può utilizzare una delle seguenti tre differenti tecniche:

1. Definire 10 variabili del tipo intero: num1, num2, …, num10, e poi calcolare la media

come: media = (num1 + num2 + …. num10)/ 10.

1. Definire una sola variabile, num, ed eseguire 10 volte, in un ciclo iterativo, l’operazione: somma = somma + num.
2. Utilizzare la struttura dati array (**il metodo migliore**).

Il primo metodo diventa inefficiente se il numero di elementi non è piccolo; il codice diventa troppo lungo e il programmatore impiega più tempo a scriverlo.

Il secondo metodo è anch’esso poco pratico in quanto utilizzando una sola variabile non siamo in grado di recuperare i dati memorizzati dato che quello nuovo sovrascrive il precedente.

Utilizziamo l’ultimo metodo per svolgere il seguente esempio:

Definire un vettore di 10 elementi del tipo carattere e, a partire dalla prima cella, assegnare le lettere maiuscole A, B, L.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| A | B | C | D | E | F | G | H | I | L |

 char vet[10];

 vet[0]='A';

 int i=1;

 do{

 vet[i]=vet[i-1];

 ++vet[i];

 i++; //conta=conta+1

 }while(i<10);

 for(int i=0; i<10; i++)

 cout<<vet[i]<<"\t";

char vet[10]; //definisco un vet di 10 elementi del tipo carattere.

vet[0]='A'; //nella posizione 0 del vet inseriamo il carattere A

vet[1]='B'; //nella posizione 0 del vet inseriamo il carattere B

………

vet[9]='L'; //nella posizione 0 del vet inseriamo il carattere L

cout<<vet[0];

//stampa a video il contenuto della cella 0 (la prima), cioè A

cout<<vet[6];

//stampa a video la G, contenuta nella cella 6 (la settima)

vet[4]='Z';

//inserisce la Z nella pos 4 del vettore (sovrascrivo la lettera E)

cin>>vet[5]; /\*il char che viene acquisito da tastiera viene salvato nella cella numero 5.\*/

**Esempio**

Come caso di applicazione di un vettore numerico consideriamo il seguente esempio:

le temperature rilevate nel corso della giornata in una determinata località sono conservate in un vettore, si richiede di scrivere un programma che, acquisita una determinata temperatura, determini se nella località considerata si è avuta nel corso della giornata tale temperatura e, in caso positivo, in quale rilevazione è registrata tale temperatura.

In termini informatici si tratta di effettuare una scansione sequenziale di un vettore per verificare se tale vettore contiene un determinato valore e, in caso positivo, se ne vuole conoscere la posizione.

Nelle righe più interessanti è presente un commento numerico per le osservazioni successive.

#include <cstdlib>

#include <iostream>

using namespace std;

int main(int argc, char \*argv[]){

 int temp[8],temprif; /\*1\*/

 int i,pos;

 /\* Acquisizione delle temperature rilevate \*/

 for(i=0;i<8;i++){

 cout<<"\nRilevazione temperatura num "<<i+1<<": ";

 cin>>temp[i];

 }

 cout<<"\nTemperatura da ricercare: ";

 cin>>temprif;

 /\* Scansione del vettore delle temperature \*/

 pos=-1; /\*2\*/

 for(i=0;i<8 && pos==-1;i++){ /\*3\*/

 if(temprif==temp[i])

 pos=i; /\*4\*/

 }

 /\* Risultati della ricerca \*/

 if(pos==-1) /\*5\*/

 cout<<"\nTemperatura non rilevata\n";

 else

 cout<<"\nTemp. acquisita nella rilevazione "<<pos+1<<endl;

 system("PAUSE");

 return EXIT\_SUCCESS;

}

* Nella riga 1 si dichiara un vettore di 8 interi per conservare le rilevazioni delle temperature e una variabile per conservare la temperatura da cercare.
* Nella riga 2 si inizializza una variabile che conterrà la posizione nella quale si troverà la temperatura cercata. Il valore -1 dipende dal fatto che le posizioni ammesse vanno dal valore 0 in su e, quindi, per indicare un valore che non corrisponda ad una posizione valida occorrerà utilizzare per esempio un numero negativo.
* Nella riga 3 inizia il ciclo di scansione del vettore alla ricerca della temperatura di riferimento. Il controllo di fine ciclo si basa su due eventi che devono verificarsi contemporaneamente: elementi non finiti (i<8) e temperatura non trovata (pos==-1). Se infatti la temperatura cercata viene ritrovata è inutile continuare ancora ad esaminare gli altri elementi del vettore. Più avanti si esaminerà un modo diverso di uscita anticipata da un ciclo a contatore.
* Se la temperatura ricercata è presente nel vettore allora, nella riga 4, se ne conserva la posizione nella variabile pos.
* La riga 5 seleziona il caso temperatura non trovata (volere di pos non modificato) dalla rilevazione ritrovata.

L’algoritmo sviluppato si basa sull’ipotesi che, se c’è, la temperatura ricercata si presenta una sola volta. Per togliere tale limitazione è necessario predisporre un vettore per contenere le posizioni delle varie ricorrenze. Le modifiche da apportare al programma saranno:

* Nella 1 dichiarare un vettore per conservare le posizioni. Per esempio postemp[8]
* Nella condizione di uscita del ciclo presente nella 3 occorre togliere il test su pos. Qui infatti se si trova la temperatura cercata occorre esaminare tutto l’array alla ricerca delle eventuali altre ricorrenze.
* La riga 4 verrebbe modificata in: postemp[++pos]=i; bisogna cioè conservare la posizione i nell’array postemp nella posizione successiva a quella indicata da pos (si tenga presente che il valore iniziale è -1). Il contatore pos va quindi *prima* incrementato e *poi* utilizzato come indice del vettore.
* Naturalmente la struttura condizionale 5 viene modificata per permettere di mandare in output l’array postemp. Si può utilizzare un ciclo a contatore simile a quello utilizzato per l’input delle temperature: l’indice i varierà fra 0 e pos e il ciclo conterrà una cout per la stampa dell’elemento dell’array postemp.

È possibile, per specificare le dimensioni di un array, usare delle costanti definite, come ad esempio:

#define ARRAY\_UNO\_MAX 10
#define ARRAY\_DUE\_MAX 20

int array\_uno[ARRAY\_UNO\_MAX];
char array\_due[ARRAY\_DUE\_MAX];

L’utilizzo di costanti per definire le dimensioni di un array è una pratica altamente consigliata. Infatti, uno degli errori che spesso accadono nella manipolazione degli elementi di un array è quello di far riferimento ad elementi che vanno oltre il limite della dimensione dell’array precedentemente definita.
Il C++ (e lo stesso C), come magari si sarebbe portati a pensare, non effettua nessun controllo sugli indici che si utilizzano quando si eseguono operazioni sugli array. Per cui, ad esempio, se eseguiamo il seguente codice:

int array\_tre[10];
……
…… // L’array viene inizializzato
int i;
for(i=0; i < 14; i++){
 cout << array\_tre[i] << endl;
}

Il risultato sarà corretto fino a quando la variabile i sarà minore o uguale a 9 (infatti la dimensione dell’array è stata definita essere uguale a 10). Quello che avverrà, quando il programma cercherà di andare a leggere i valori array\_tre[10], array\_tre[11], array\_tre[12] e array\_tre[13] è assolutamente casuale. Infatti, il programma andrà a leggere delle aree di memoria che non sono state inizializzate o, addirittura, che contengono dei valori relativi ad altre variabili, stampando quindi sullo schermo risultati inaspettati. Se si fosse usata una costante per definire la dimensione dell’array questo errore sarebbe stato facilmente evitato. Infatti:

#define ARRAY\_trE\_MAX 10

int array\_tre[ARRAY\_trE\_MAX];
……
…… // L’array viene inizializzato
int i;
for (i=0; i < ARRAY\_trE\_MAX; i++){

cout << array\_tre[i] << endl;
}

Come si può notare facilmente in questo modo il programmatore non deve preoccuparsi di ricordarsi la dimensione dell’array array\_tre poiché la costante ARRAY\_trE\_MAX viene utilizzata proprio per tale scopo, evitando i cosiddetti errori di “**array out of bounds**”.

# STRINGHE

I dati di tipo alfanumerico formati da più caratteri si chiamano **stringhe** e possono essere considerati come una sequenza di dati di tipo char.

Il C, contrariamente ad altri linguaggi di programmazione, non mette a disposizione del programmatore il tipo stringa. Per identificare, quindi, una stringa sarà necessario definire un array di caratteri. Una stringa è vista come una sequenza di caratteri (array di caratteri) terminata da: \0, contenuta in uno spazio in memoria (**buffer**).

Vediamo un semplice programma che mostra l’uso delle stringhe.

#include <cstdlib>

#include <iostream>

using namespace std;

int main(int argc, char \*argv[]){

 char stringa1[6];

 char stringa2[6];

 // Inizializzazione durante la definizione

 char stringa3[5] = "nave";

 // Inizializzazione della stringa stringa1

 stringa1[0] = 'p';

 stringa1[1] = 'a';

 stringa1[2] = 'l';

 stringa1[3] = 'l';

 stringa1[4] = 'a';

 stringa1[5] = '\0'; //carattere terminatore stringa

 // Valorizzazione effettuata da STDIN (tastiera)

 cout<<"Scrivi una parola: ";

 cin >> stringa2;

 cout << "Stampo le tre stringhe\n";

 cout << stringa1;

 cout << "\n";

 cout << stringa2;

 cout << "\n";

 cout << stringa3;

 cout << "\n";

 system("PAUSE");

 return EXIT\_SUCCESS;

}

L’output del programma

palla

pippo (Supponendo che l'utente abbia inserito la stringa pippo)

nave

**LA LIBRERIA STRING**

Una stringa è semplicemente un array di caratteri, ovvero una sequenza di caratteri terminante con uno zero binario (carattere terminatore della stringa: \0).

La gestione delle stringhe in C++ è semplificato da una serie di funzioni richiamabili dalla libreria string.

Questo tipo di dato consente la semplificazione della gestione dei dati alfanumerici.

Vediamo quali sono le tipiche operazioni che si possono fare su una stringa:

* #include <string>: libreria da richiamare per utilizzare le stringhe
* string stringa: dichiarazione di una stringa
* string stringa="Daniele Corti": dichiarazione e inizializzazione di una stringa
* string stringa="Daniele\nCorti": multiriga
* stringa.length(): restituisce la lunghezza della stringa
* if(nomeStringa.empty()): restituisce TRUE se la stringa è vuota, altrimenti FALSE
* stringa3=stringa1 + stringa2: concatenazione di stringhe con l’operatore (+)
* if(stringa1 ==stringa2): restituisce TRUE se le due stringhe sono uguali
* cout<<stringa1[i]: accesso alla i-esimo carattere della stringa
* stringa2.insert(n, stringa1): inserimento della stringa1 nella n-esima posizione della stringa2
* stringa1.substr(start, numCaratteri): estrazione da stringa1 di nCaratteri a partire da start carattere
* stringa1.erase(start, numCaratteri): cancellazione di numCaratteri a partire da start caratteri
* stringa2.replace(start, numCaratteri, stringa1): a partire da start carattere inserisco numCaratteri di stringa1 in stringa2
* stringa.find(StringaDaCercare, start): restituisce la posizione del primo carattere della StringaDaCercare in stringa a partire da start esimo carattere.
* cin>>stringa1: input stringa senza spazi
* cin.getline(stringa1, numeroCaratteri): input stringa con spazi

#include <cstdlib>

#include <iostream>

#include <string>

using namespace std;

int main(int argc, char \*argv[]){

 string s1, s2="Ciao", s3="Daniele", s4;

 cout<<s2<<endl;

 s2="Daniele\nCorti"; //multiriga

 cout<<s2<<endl;

 //restituisce la lunghezza della stringa

 int len=s2.length();

 cout<<"La stringa ha "<<len<<" caratteri "<<endl;

 cout<<"(compresi gli spazi)"<<endl;

 if(s1.empty()) //verifico se la stringa è vuota

 cout<<"Lettura fallita!"<<endl;

 //concateno le stringhe con l'operatore +

 s4=s2 + " " + s3;

 cout<<s4<<endl;

 s4=s2 + " " + s3 + '\n'; //e vado a capo

 cout<<s4;

 //confronto fra stringhe

 if(s2==s3)

 cout<<"Stringhe uguali\n";

 else

 cout<<"Stringhe diverse\n";

 //accesso agli elementi della stringa tramite l'operatore []

 //stampo il primo carattere della stringa s2

 cout<<s2[0]<<endl;

 //inserire una stringa in un'altra

 string sa="Vittorio Secondo";

 string sb="Emanuele ";

 sa.insert(9,sb); //sa diviene "Vittorio Emanuele Secondo"

 cout<<sa<<endl;

 //estrazione di una sottostringa

 string frase="Daniele Corti";

 cout<<frase.substr(8,5)<<endl; //Corti

 //estrae dalla frase la parola Corti (inizio, quanti)

 //cancellazione di una sottostringa

 string frase2="Cara amica ti scrivo\n";

 frase2.erase(5,6);

 cout<<frase2; // "caro ti scrivo"

 //cancello e sostituisco

 frase2.replace(0,4,"Anna");

 cout<<frase2<<endl; //Anna ti scrivo

 //ricerca sottostringa

 string nome="Maria Silvia";

 int startPos=nome.find("Silvia", 0);

 cout<<startPos<<endl;

 system("PAUSE");

 return EXIT\_SUCCESS;

}

**GESTIONE DELLE PAROLE SEPARATE DALLO SPAZIO**

Sappiamo che il tipo string non gestisce parole separate dallo spazio, ovvero se scriviamo questo:

string frase;

cin>>frase;

e l’utente inserire una sequenza di caratteri contenenti spazi, nella variabile frase verrà salvata solo la prima parte della stringa preceduta dal primo spazio digitato.

Per risolvere questo problema possiamo utilizzare, per le variabili vettori di caratteri, la funzione:

cin.getline(nomeVariabile, numeroCaratteri)

richiamabile dalla libreria #include <iostream>

Esempio:

 char frase[20];

 //lunghezza massima della frase. 20 caratteri

 cout<<"Scrivi una frase\n";

 cin.getline(frase, 20);

Per le variabili del tipo string invece esiste la funzione:

getline(cin, variabileStringa);

Esempio:

 string frase;

 getline(cin, frase);

 cout<<frase;

# ARRAY PARALLELI

Gli array paralleli sono 2 o più array che hanno lo stesso indice e contengono elementi in relazione fra loro anche di tipologia differente.

Gli array paralleli permettono di gestire quindi tabelle di dati eterogenei. Per esempio:

int size=6;

string nome[size];

int eta[size];

nome[0]= "Paolo"; eta[0]=22;

nome[1]= "Luca"; eta[1]=18;

nome[2]= "Elisa"; eta[3]=17;

nome[3]= "Marco"; eta[3]=23;

nome[4]= "Anna"; eta[4]=15;

nome[5]= "Luisa"; eta[5]=25;

nome[6]= "Davide"; eta[5]=13;

|  |  |  |
| --- | --- | --- |
| indice  | nome | eta |
| 0 | Paolo | 22 |
| 1 | Luca | 18 |
| 2 | Elisa | 17 |
| 3 | Marco | 23 |
| 4 | Anna | 15 |
| 5 | Luisa | 25 |
| 6 | Davide | 13 |

* Il primo array contiene stringhe di nomi di persone.
* Il secondo array contiene interi che rappresentano le rispettive età delle persone del primo array.

**Esercizio 1**

Si vogliono gestire le prenotazioni delle visite in ospedale da parte degli utenti. Per ogni prenotazione siamo interessati al nominativo dell’utente (nome e cognome), alla visita (tipo di visita prenotata), alla data (giorno, mese, anno) e all’ora di prenotazione.

1. Creare la seguente tabella in C++ utilizzando gli array paralleli:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Nominativo | Visita | Data | Ora |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| … |  |  |  |  |
| 9 |  |  |  |  |

int DIM=10; //10 righe

string nominativo[DIM];

1. Creare lo script che consenta all’operatore di inserire i dati relativi alle DIM prenotazioni nella tabella.
2. Visualizzare i nominativi che hanno prenotato una certa visita (visita è il parametro di ricerca digitabile dall’operatore).
3. Visualizzare il numero di visite prenotate da un certo nominativo.
4. Visualizzare il numero di visite prenotate in un determinato giorno.
5. Visualizzare le visite (e i nominativi) prenotate in un determinato giorno.
6. Come è possibile modificare il programma per consentire all’operatore di aggiungere una nuova prenotazione.

**Esercizio 2**

I dati relativi alle vendite di una giornata nei 5 reparti di un supermercato sono memorizzati in quattro vettori paralleli:

* Articoli (nome dell’articolo venduto),
* Reparto (nome del reparto relativo all’articolo venduto),
* Valore (numero di quantità vendute di quell’articolo).
* Prezzo (prezzo unitario dell’articolo).

Scrivere un programma che visualizzi i dati relativi alle vendite di ogni reparto e alla sua percentuale rispetto al totale.

**Esercizio 3**

I dati relativi al campionato di calcio sono memorizzati in tre vettori paralleli:

* Squadre (nome della squadra).
* Serie ("a" o "b").
* Punti (punteggio ottenuto nel campionato).

Scrivere un programma con le seguenti richieste:

1. stampare un tabulato contenente lelenco delle squadre di serie a nella forma:

**elenco \_squadre\_serie\_a**

squadra punti

......... .......

1. Dato in input il nome della squadra, stampare la serie e i punti realizzati. Se la squadra non esiste stampare un messaggio di errore.
2. Costruire e stampare un altro vettore classifb contenente solo le squadre di serie b.
3. Cercare e stampare il nome della squadra campione di serie a.

# ARRAY MULTIDIMENSIONALI

Come abbiamo visto, il termine dimensione rappresenta il numero di indici utilizzati per far riferimento a un determinato elemento dell’array. Gli array che abbiamo visto finora erano monodimensionali e richiedevano l’uso di un solo indice. E’ facile determinare il numero di dimensioni di un array osservando la sua dichiarazione. Se nella dichiarazione è presente una sola coppia di parentesi quadre ([ ]) l’array è monodimensionale. Due coppie di parentesi quadre ([ ] [ ]) indicano, invece, un array bidimensionale, e così via. Il numero massimo di dimensioni normalmente solitamente non supera 3. Per immaginare la rappresentazione di un array bidimensionale si può far riferimento ad una tabella (o a una matrice) in cui il primo indice dell’array rappresenta la riga ed il secondo rappresenta la colonna.

# LE MATRICI

Il caso più semplice di array multimensionali è quello di dimensione 2, le **matrici**.

Vediamo un esempio di dichiarazione:

int matrix[4][5];

In questo esempio si dichiara una matrice di interi di dimensione 4x5 (quattro righe e cinque colonne).



Per accedere ad un singolo elemento di un array bidimensionale usando doppi indici (uno per le righe e l’altro per le colonne).

![matrix[3][1]]()

Per esempio, per selezionare l’elemento della quarta riga – seconda colonna:

matrix[3][1];

NB Ricordate che gli indici partono da 0 e, quindi, la quarta riga ha indice 3 e la seconda colonna ha indice 2.

Vediamo quest’altro esempio:

int a[3][4];



Analogamente agli array monodimensionali, anche gli array multidimensionali possono essere inizializzati in fase di dichiarazione.

char tris[3][3] = {

 { 'X', '0', '0' },

 { '0', 'X', '0' },

 { '0', 'X', 'X' }

};

Spesso è conveniente usare dei cicli annidati per inizializzare / accedere gli elementi di un array multidimensionale.

int matrix[4][4];

for(int riga=0; riga<4; riga++){

 for(int colonna=0; colonna <4; colonna ++){

 matrix[riga][ colonna] = 0

 }

}

L’uso di cicli annidati consente di scrivere in maniera naturale gli algoritmi che operano su matrici (o più in generale, che operano su array multidimensionali). Ad esempio, vediamo come scrivere un programma che calcola la somma di tutti gli elementi di una matrice e stampa il risultato:

int a[4][4] = {

            {1, 2, 3, 4, 5},

            {4, 8, 3, 2, 7},

            {10, 7, 3, 6, 8},

            {9, 2, 8, 3, 1}

};

int somma=0;

for (int i=0; i< a.length; i++)

    for (int j=0; j<a[i].length; j++)

         somma = somma + a[i][j];

**PROCEDURE – FUNZIONI**

Un problema, se complesso, può essere scomposto in sotto-problemi più semplici. Ogni sotto-problema potrà così essere risolto da un singolo sotto-programma (procedura o funzione) indipendentemente dagli altri. Tutti i sotto-programmi, assieme al sotto-programma principale MAIN, che deve essere sempre presente, collaborano quindi fra loro e, se necessario, si scambiano informazioni, al fine di far fare al programma quello richiesto dalle specifiche del problema da risolvere.



L’esecuzione di un programma inizia sempre dalla prima istruzione contenuta nella funzione MAIN; questa può chiamare altre funzioni, che a loro volta ne possono chiamare altre ancora. L’unico limite è rappresentato dalla quantità di memoria disponibile.

In generale, è utile suddividere il programma in parti bene definite, e codificare ciascuna di esse mediante un sotto-programma dedicato; ciò può rivelarsi particolarmente opportuno soprattutto per quelle parti di elaborazione che devono essere ripetute più volte su dati differenti.

La procedura, infatti, può essere richiamata dalle altre più volte: **riutilizzo del codice**. Lo stesso codice può anche essere riutilizzato in altri contesti adattandolo se necessario alle specifiche del problema da risolvere.

Il riutilizzo del codice non è però l’unico criterio che conduce ad individuare porzioni di codice atte ad essere racchiuse in funzioni: l’importante, come si è accennato, è isolare compiti logicamente indipendenti dal resto del programma; è infatti usuale, in C++, definire funzioni che nel corso dell’esecuzione vengono chiamate una volta sola. Di solito si fa in modo che una funzione svolga un unico compito, almeno quando occorre svolgere nuovamente quel dato compito basta richiamare la funzione.

Fino ad ora i programmi sono stati visti come una sequenza di istruzioni all’interno della funzione MAIN. Il MAIN può anche chiamate funzioni esterne, passandogli degli argomenti.

**Esempio**

Utilizziamo la funzione predefinita POW, contenuta nella libreria <cmath>, e determiniamo il risultato dell’operazione baseesponente dove, base ed esponente sono i dati di input.

NB la funzione pow, utilizzabile grazie alla inclusione della libreria cmath, accetta due parametri d’ingresso, il primo è la base e il secondo l’esponente, e restituisce il risultato dell’operazione baseesponente.

|  |  |
| --- | --- |
| **NUM** | **CODICE** |
| 12345678910111213 | #include <iostream>#include <cmath>> //libreria matematicheusing namespace std;int main(int argc, char\*\* argv){ int risultato, base, esponente; cout<<"Base: "; cin>>base; cout<<"Exp: "; cin>>esponente; risultato=pow(base, esponente); //funzione potenza cout<<risultato; return 0;} |

Alla riga 11 viene chiamata la funzione (subroutine o sottoprogramma) predefinita **pow** che si occupa di restituire il risultato del calcolo baseesponente, dove base ed esponente sono i parametri passategli come argomenti. Il risultato restituito viene assegnato alla variabile risultato.

Le procedure si comportano allo stesso modo delle funzioni matematiche:

y = f(x1, x2, …, xn)

**Punti di vista**

Quando si studia una funzione occorre distinguere con chiarezza la definizione dall’invocazione.

Per fare una metafora, la funzione va pensata come un soggetto al quale un altro soggetto (il MAIN) gli dà l’incarico di svolgere un determinato compito; il soggetto dopo aver concluso il compito assegnatoli restituisce al soggetto MAIN un risultato.

Proviamo a pensare alla seguente situazione: “revisione veicolo”. Ogni cittadino deve portare il proprio veicolo ad un’officina meccanica autorizzata per essere revisionato. Se il veicolo supera una serie di test può continuare a circolare per altri due anni altrimenti deve essere rottamato. L’officina meccanica risponde, quindi, con SI, se il veicolo ha superato la revisione e, NO in caso contrario. L’officina meccanica incarica un suo meccanico specializzato per svolgere il compito di verifica del veicolo: quando gli viene passato il veicolo, egli lo valuta e restituisce al cliente un documento in cui si autorizza (SI) il veicolo alla circolazione per altri due anni oppure meno (NO).

In questa metafora:

* il meccanico è la **funzione**.
* il cittadino che porta il veicolo all’officina meccanica per essere revisionato, è il **chiamante**.
* il meccanico, per poter svolgere il suo compito, deve ricevere un veicolo da revisionare; il veicolo rappresenta un **parametro**. Il parametro è l’oggetto che gli viene passato e su cui deve lavorare: senza parametro non può lavorare.
* l’autorizzazione alla circolazione rappresenta il risultato atteso: in pratica ciò che la funzione **restituisce** al chiamante.
* quando un cittadino presenta la richiesta di revisione alla officina meccanica sta chiamando (**invocazione**) la funzione.
* quando l’officina meccanica spiega al suo meccanico specializzato cosa deve fare, ovvero lo istruisce sul compito da svolgere, è il momento di **definizione della funzione**.

**Definizione**

Abbiamo visto come utilizzare le funzioni predefinite in C++. Certamente, in C++, è possibile non solo di utilizzare funzioni già esistenti ma anche di crearne nuove.

La definizione è il momento in cui il programmatore stabilisce il comportamento della funzione. Di solito questo momento viene accorpato a quello della dichiarazione, che corrisponde al momento in cui stabilisco un nome per la funzione, stabilisco il tipo della funzione e decido il numero ed il tipo dei parametri della funzione.

La sintassi della definizione di una funzione è la seguente:

tipoRestituito nomeFunzione(tipo1 nomePar1, tipo2 nomePar2, …, tipoN nomeParN){

 <corpo della funzione>;

 return(nomeVarDaRestituire);

}

* **tipoRestituito** indica il tipo di valore (int, float, string, etc.) che la funzione restituisce attraverso l’istruzione return. Se la funzione non restituisce nessun valore deve essere dichiarata di tipo void e va omessa l’istruzione return. Per esempio: una funzione che rende il massimo tra due interi deve essere di tipo intero; una funzione che prende due variabili e scambia il contenuto non rende alcun dato ed è di tipo void. Una funzione in C++ può restituire sia tipi valore (come int, bool, double, strutture) oppure string (string) oppure riferimento (int[], oggetti) come vedremo.
* **nomeFunzione** indica l’identificatore della funzione ovvero il nome che scegliamo per la funzione e servirà per invocarla in seguito.
* **nomePar1** è il nome del primo parametro (variabile), **parametro formale**, passato alla funzione (argomento di ENTRATA della funzione).
* **Tipo1** è il tipo del primo parametro (variabile).
* **nomePar2**, …., **nomeParN** in modo analogo sono gli altri argomenti d’ingresso.
* **Tipo2**, …, **TipoN** in modo analogo sono i relativi tipi.

Una funzione può richiedere uno o più parametri in ingresso, ma anche nessuno. In quest’ultimo caso occorre omettere i parametri oppure dichiararli di tipo void.

* <corpo della funzione> Il corpo di una funzione è una normale sequenza di dichiarazioni di variabili (LOCALI), di istruzioni, di chiamate a funzione che definiscono le operazioni che la funzione esegue quando viene invocata, l'unica cosa che esso non può contenere è un’altra definizione di funzione. Questo perché tutte le funzioni hanno pari livello gerarchico e quindi non possono essere nidificate, cioè definite l’una all'interno di un'altra. L’esecuzione della funzione termina quando si incontra l’ultima istruzione presente nel corpo oppure l’istruzione return: in entrambi i casi l’esecuzione ritorna alla funzione chiamante.

Occorre però soffermarsi brevemente sull’istruzione return.

* Se la funzione non è dichiarata void è obbligatorio utilizzare la return per uscire dalla funzione (anche quando ciò avvenga al termine del corpo), in quanto essa rappresenta l'unico strumento che consente di restituire un valore alla funzione chiamante.  Detto valore, indicato tra parentesi tonde dopo return, può essere una costante, una variabile o, in generale, un'espressione (anche una chiamata a funzione).  E' ovvio che il tipo del valore specificato deve essere il medesimo restituito dalla funzione.
* Se invece la funzione è dichiarata void, e quindi non restituisce alcun valore, l'uso dell'istruzione return è necessario solo se l'uscita deve avvenire (ad esempio in dipendenza dal verificarsi di certe condizioni) prima della fine del corpo (tuttavia non è vietato che l'ultima istruzione della funzione sia proprio una return). In tal caso a destra della return non deve essere specificato alcun valore, bensì direttamente il solito punto e virgola.
* All'interno di una funzione si possono mettere anche più return, ciò accade quando si ha la necessità di restituire valori diversi.

A tale proposito si ricorda che main() essendo una funzione, senza argomenti (parametri e valore restituito), può essere dichiarata senza specificare il tipo di ritorno ed il tipo dei parametri, e nel corpo può essere omesso return.

Perché una funzione possa essere chiamata è necessario, che essa sia definita prima della riga di codice che la richiama.

#include <iostream>

using namespace std;

int somma(int x,int y){

 return(x+y);

}

/\* Funzione principale, eseguita per prima \*/

int main(int argc, char\*\* argv){

/\* Inizio della funzione main() \*/

 int a,b,c; /\* Definiamo 3 variabili intere \*/

 cout<<"Dammi un numero: ";

 cin>>a;

 cout<<"Dammi un altro numero: ";

 cin>>b;

 c = somma(a,b); /\* Chiamata della funzione!

 a,b parametri attuali\*/

 cout<<"La somma dei due numeri e' "<<c<<endl;

 return(0); /\* la funzione main restituisce uno 0 intero \*/

} /\* Fine della funzione main() \*/

In alternativa, il compilatore deve conoscerne le regole di chiamata (parametri e valore restituito): pertanto è necessario, inserire prima della funzione main il solo prototipo della funzione stessa: con tale termine si indica la prima riga della definizione senza gli identificatori degli argomenti (e dopo il main la funzione per esteso).

Esempio:

#include <iostream>

using namespace std;

/\* Dichiarazione anticipata (prototipo) della funzione \*/

int somma(int,int);

/\* Funzione principale, eseguita per prima \*/

int main(int argc, char\*\* argv){

 /\* Inizio della funzione main() \*/

 int a,b,c; /\* Definiamo 3 variabili intere \*/

 cout<<"Dammi un numero: ";

 cin>>a;

 cout<<"Dammi un altro numero: ";

 cin>>b;

 c = somma(a,b); /\* Chiamata della funzione!

 a,b parametri attuali\*/

 cout<<"La somma dei due numeri e' "<<c<<endl;

 return(0); /\* la funzione main restituisce uno 0 intero \*/

} /\* Fine della funzione main() \*/

int somma(int x,int y){

 return(x+y);

}

Tale dichiarazione è sufficiente al compilatore per verificare che le chiamate a somma(a,b) siano eseguite correttamente.

I prototipi sono inoltre l’unico strumento disponibile per comunicare al compilatore i nomi delle funzioni di libreria richiamate nei sorgenti: infatti, essendo disponibili sotto forma di codice oggetto precompilato, esse non vengono mai definite.

**Parametri formali e attuali**

Le variabili x e y dichiarate nella definizione della funzione si chiamano **parametri formali**, mentre le variabili a e b utilizzate nel main per chiamare la funzione somma si chiamano **parametri attuali**.

 I parametri formali x, y sono variabili LOCALI, ovvero utilizzabili solo dalla funzione, ed assumeranno i valori passati con "somma(a, b)": ciò significa che il valore di a sarà copiato in x, quello di b in y e la loro somma sarà data in uscita, e copiata in c.

E’ estremamente importante osservare che una funzione non può mai modificare i parametri attuali che le sono passati, in quanto ciò che essa riceve è in realtà una copia dei medesimi.

Quando il controllo passa dalla funzione chiamante alla funzione chiamata vengono effettuate automaticamente le seguenti operazioni:

* creazione delle variabili locali della funzione chiamata (compresi ovviamente i parametri formali),
* copia, rispettando l’ordine, dei parametri attuali (della funzione chiamante) nei corrispondenti parametri formali (della funzione chiamata)

Al termine dell’esecuzione il valore calcolato dalla funzione viene trasmesso alla funzione chiamante, attraverso return e le variabili locali della funzione chiamata vengono rilasciate (scompaiono).

All’interno di una funzione si possono anche definire anche delle variabili LOCALI, ossia utilizzabili solo all'interno della funzione.

Esempio:

int somma(int x, int y){
     int risultato; //Definisco una variabile locale
     risultato=x+y;
     return(risultato);
  }

Generalmente una funzione richiede dei parametri in ingresso e opzionalmente produce un risultato in uscita ma, comunque, certe funzioni possono non richiedere dati in ingresso e/o non restituire alcun valore in uscita, ad esempio:

* una funzione che non restituisce nessun valore, ma richiede dati di ingresso, potrebbe essere una funzione dedicata alla stampa (come ad esempio scanf, che in entrata vuole il testo da stampare e in output non produce dati);
* una funzione che non richiede dati in ingresso e che non restituisce dati in uscita potrebbe essere una funzione che pulisce lo schermo;
* una funzione che non richiede dati in ingresso e fornisce dati in uscita potrebbe essere una routine che genera numeri casuali.

Funzione che non ha parametri né in entrata né in uscita:

 void acapo(){
   cout<"\n";

/\* se una funzione non restituisce alcun valore, si può

omettere il return \*/
}
Funzione che ha solo un parametro in entrata:

void stampanumero(int a){
   cout<<a;

/\* se una funzione non restituisce alcun valore, si può

omettere il return\*/

}

Funzione che ha solo un parametro in uscita:
int legginumero(){
   int a;  /\* serve una variabile locale...\*/
   cin>>a;
   return(a);
}

**ESEMPIO**//funzione senza parametri
static public int Rende22(){
 return 22;
}

// funzione con 1 parametro
int Quadrato(int x){
 return x\*x;
}

// funzione con 2 parametri
int Massimo(int x, int y){
 if(x > y)
 return x;
 else

return y;
}

Esempio:

#include <iostream>

using namespace std;

//Dichiarazione anticipata (prototipo) delle funzioni

int somma(int, int);
void acapo();
void stampanumero(int);
int legginumero();

//Funzione principale e inizio del programma

int main(int argc, char\*\* argv){
 int a,b,c; /\* Definiamo 3 variabili intere \*/

 acapo();
 cout<<"Dammi un numero: ";
 a = legginumero();
 acapo();
 cout<<"Dammi un altro numero: ";
 b = legginumero();
 c = somma(a,b);
 acapo();
 cout<<"La somma dei due numeri e’: ";
 acapo();
 stampanumero(c);
}

int somma(int a, int b){
 return(a+b);
}

void acapo(){
  cout<<"\n"; /\* se una funzione non restituisce
                alcun valore, si può omettere il return\*/
}

void stampanumero(int a){
  cout<<a; /\* se una funzione non restituisce
                alcun valore, si può omettere il return\*/

}

int legginumero(void){
  int a;  /\* serve una variabile locale...\*/
  cin>>a;
  return(a);
}

Si noti il diverso uso tra

* le funzioni che restituiscono un valore e
* le funzioni di tipo void (che non restituiscono niente).

Le prime si chiamano per calcolare un valore (il valore da loro restituito), pertanto saranno presenti in espressioni, istruzioni di assegnamento o di stampa, le seconde invece poiché non generano un valore in output corrispondono alla chiamata di sottoprogrammi (in altri linguaggi vengono chiamate procedure o subroutine).

La chiamata di una funzione che restituisce un valore di ritorno può essere inserita come operando in qualsiasi espressione o come argomento nella chiamata di un’altra funzione (previo controllo da parte del compilatore che il tipo della funzione sia ammissibile): la chiamata viene eseguita con precedenza rispetto alle altre operazioni e al suo posto viene sostituito il valore di ritorno restituito dalla funzione.

Se invece la funzione è di tipo void (oppure il suo valore di ritorno non interessa) la chiamata deve assumere la forma di un’istruzione a sé stante.

Usare le funzioni in un programma così corto può sembrare controproducente, ma più il programma diventa grande e più le funzioni divengono utili.

Osservazioni

* Per il compilatore C++, un identificatore seguito da una coppia di parentesi tonde è sempre una chiamata a funzione.
* Dal punto di vista elaborativo la chiamata di una funzione corrisponde al trasferimento dell'esecuzione dalla funzione chiamante, al blocco di codice che costituisce la funzione chiamata; la funzione chiamante deve conoscere della funzione chiamata, esclusivamente il tipo di parametri che essa si aspetta in ingresso e, se previsto, il tipo di valore (uno ed uno solo) di ritorno.  Tale valore può essere considerato il risultato di un'espressione e come tale, lo si può passare ad un'altra funzione o memorizzarlo in una variabile, ma può anche essere ignorato.
* Quando una funzione ne chiama un'altra, il controllo dell'esecuzione passa a quest'ultima che, al termine del proprio codice, o in corrispondenza dell'istruzione return lo restituisce alla funzione chiamante.
* Ogni funzione può chiamare anche se stessa, secondo una tecnica detta ricorsione.
* Come detto in precedenza una funzione non può mai modificare i parametri attuali che le sono passati, in quanto può accedere solo alle variabili locali, infatti il passaggio dei parametri alla funzioni avviene per valore e non per riferimento, ovvero non viene passato il suo indirizzo, bensì il suo valore (i parametri formali contengono la copia dei parametri attuali).
* Grazie a questo meccanismo di protezione le funzioni possono dunque modificare a piacere i parametri ricevuti senza il rischio di modificare i dati della funzione chiamante.
* L'impossibilità, per ciascuna funzione, di accedere a dati non locali ne accentua l'indipendenza da ogni altra parte del programma.
* Una eccezione è rappresentata dalle variabili globali, visibili per tutta la durata del programma e accessibili in qualsiasi funzione.
* Vi è poi una seconda eccezione: i puntatori. A dire il vero essi sono un'eccezione solo in apparenza, ma di fatto consentono comportamenti contrari alla regola, appena enunciata, di inaccessibilità a dati non locali.
* Quando un puntatore è parametro formale di una funzione, il parametro attuale corrispondente rappresenta l'indirizzo di un'area di memoria. Coerentemente con quanto affermato, alla funzione chiamata è passata una copia del puntatore, salvaguardando il parametro attuale, ma tramite l'indirizzo contenuto nel puntatore la funzione si può accedere all'area di memoria "originale", in quanto, è bene sottolinearlo, solo il puntatore viene duplicato, e non l'area di RAM referenziata. E' proprio tramite questa apparente incongruenza che le funzioni possono modificare le variabili di cui ricevano, quale parametro, l'indirizzo (o meglio, il puntatore).
* A differenza di altri linguaggi il C++ non ha procedure o subroutine poiché usa le funzioni per soddisfare entrambe le esigenze.

**APPROFONDIMENTO Trasmissione dei parametri tramite l’area stack**

Nei problemi di programmazione è sentita la necessità di gestire due particolari categorie di liste caratterizzate da metodi di accesso ben definiti e utilizzate in numerose circostanze:

* le liste di tipo queue (coda), accessibili con il metodo FIFO (first in-first out): il primo dato che entra nella lista è il primo a essere servito; tipiche queues sono le code davanti agli sportelli, le code di stampa (priorità a parte) ecc...
* le liste di tipo stack (pila), accessibili con il metodo LIFO (last in-first out): l’ultimo dato che entra nella lista è il primo a essere servito.

Nella trasmissione dei parametri fra funzione chiamante e funzione chiamata vengono utilizzate liste di tipo stack.

Quando  la funzione main viene eseguita  le sue variabili locali vengono allocate in un’area di memoria, detta appunto stack, quando questa chiama un'altra funzione (B) vengono allocati sullo stack  un insieme di dati comprendenti: le variabili locali, gli argomenti (parametri formali della funzione B) e l’indirizzo di rientro nel main  necessari alla funzione B, se anche B a sua volta chiama un’altra funzione C, sistema nell’area stack il proprio insieme di dati  “impilato” sopra il precedente.

Esempio:

Argomento 1 passato da B
Variabili locali a C
Indirizzo di rientro in B                               La funzione B chiama la funzione C con un argomento
------------------------------
Parametro 1 passato a C
Argomento 1 passato da A
Argomento 2 passato da A
Variabili locali a B
Indirizzo di rientro in A                             La funzione A chiama la funzione  B con due argomenti
------------------------------
Parametro 1 passato a B
Parametro 2 passato a B
Variabili locali ad A
------------------------------

Si noti che durante l'esecuzione il programma fa riferimento all’ultimo insieme di dati entrato nello stack, quando una funzione termina, legge l'indirizzo di ritorno e rimuove il gruppo di dati dallo stack.

Nell’esempio considerato quando C imposta l’indirizzo di rientro in B rimuove il gruppo di dati dallo stack. La stessa cosa succede quando il controllo rientra da B nel main: finché quando termina il main lo stack rimane vuoto.

**APPROFONDIMENTO Ricorsione delle funzioni**

Tornando all’esempio precedente, la trasmissione dei parametri attraverso lo stack garantisce che il meccanismo funzioni comunque, sia che B e C siano funzioni diverse, sia che si tratti della stessa funzione (ogni volta va a cercare nello stack l’indirizzo di rientro nella funzione chiamante e quindi non cambia nulla se tale indirizzo si trova all’interno della stessa funzione).

Ne consegue che in C++ le funzioni possono chiamare se stesse (ricorsività delle funzioni). *Ovviamente tali funzioni devono sempre contenere un’istruzione di controllo che, se si verificano certe condizioni, ha il compito di interrompere la successione delle chiamate.*

Esempio tipico di una funzione chiamata ricorsivamente è quello del calcolo del fattoriale di un numero intero:

int fatt (int n){

if (n<=1)  // istr. di controllo

return (1);
else
 return (n\*fatt(n-1));
}

alternativamente, cioè senza usare la ricorsione, si produce codice meno compatto:

int fatt(int n){
 int i=2, m=1;
 while(i<=n)

m\*=i++;
return m;

}

La funzione ricorsiva fatt()non necessita di iterazioni, ma include internamente il calcolo del risultato, infatti restituisce 1 se il parametro ricevuto è minore o uguale di 1 (cioè vale 0 o 1), mentre in caso contrario il valore restituito è il prodotto di n per il fattoriale di n-1, pertanto fatt() calcola il valore da restituire chiamando se stessa e "passandosi" come parametro il parametro appena ricevuto, diminuito di uno.

Il termine "chiamare sé stessa" implica una nuova istanza della funzione, ovvero la generazione nella memoria stack di una nuova area di lavoro.

La nuova istanza della funzione alloca le proprie variabili locali, ignorando l'esistenza di quelle dell'istanza precedente, grazie a questo meccanismo la funzione ha un proprio spazio "riservato" in cui operare.

A volte, però, può essere utile che una istanza conosca qualcosa delle altre: ad esempio un contatore, che consenta di sapere in qualunque istanza a che profondità si sia spinta la ricorsione.

Tale esigenza è soddisfatta dalle variabili static, in quanto esse sono locali alla funzione in cui sono definite, ma comuni a tutte le sue istanze.

Tale dichiarazione è comprensibile se si tiene conto che le variabili statiche sono accessibili solo alla funzione in cui sono definite, ma esistono e conservano il loro valore per tutta la durata del programma.

Infatti quando una funzione è chiamata per la prima volta ed assegna un valore ad una variabile statica, questa mantiene il proprio valore anche in una seconda istanza (e in tutte le successive) della stessa funzione; mentre delle variabili locali è generata una nuova copia in ogni istanza, una variabile static è unica in tutte le istanze e poiché essa esiste e mantiene il proprio valore anche in uscita dalla funzione, ogni istanza può conoscere non solo il valore che tale variabile aveva nell'istanza precedente, ma anche nell'istanza successiva (ovviamente dopo il termine di questa).

Le variabili globali, infine, sono accessibili a tutte le istanze ma, a differenza di quelle statiche, lo sono anche alle altre funzioni.

#include <stdio.h>

void incrementa(){

static int n=10;
n++;
cout<<"valore di n = "<<n;
return;

}

main(){

for(int i=0; i<3;i++)

incrementa();

}

Il programma darà in esecuzione:
valore di n = 11
valore di n = 12
valore di n = 13

In C++ la ricorsione è usata raramente.

Il punto debole dell'approccio ricorsivo alla definizione di un algoritmo consiste in un utilizzo dello stack più pesante rispetto alla soluzione iterativa, infatti ogni variabile locale definita in una funzione ricorsiva è duplicata nello stack per ogni istanza attiva.

E' perciò necessario, onde evitare disastrosi problemi in fase di esecuzione, contenere il numero delle variabili locali (soprattutto se ingombranti es: array), o richiedere al compilatore la generazione di uno stack di maggiori dimensioni.

Per i motivi precedenti è decisamente sconsigliabile definire array nelle funzioni ricorsive.
(Ad essi, come vedremo successivamente, può essere sostituito un puntatore, che comporta meno occupazione di spazio in termini di stack).

Esempio di funzione ricorsiva:

#include <stdio.h>
int fatt (int);
int main(void){

int num;

printf("\nDammi il numero  ");
scanf ("%d",&num);
printf("fattoriale di %d = %d\n",num,fatt(num));

}

int fatt (int n){
 if (n<=1)  return (1);
 else return (n\*fatt(n-1));
}

**Approfondimento Visibilità e tempo di vita di una variabile**

**Ambito di azione**

Abbiamo visto che, in via del tutto generale, si definisce ambito di azione (o ambito di visibilità o
scope) l’insieme di istruzioni di programma comprese fra due parentesi graffe: {....}

Le istruzioni di una funzione devono essere comprese tutte nello stesso ambito; ciò non esclude che si possano definire più ambiti innestati l’uno dentro l’altro (ovviamente il numero di parentesi chiuse
deve bilanciare quello di parentesi aperte, e ogni parentesi chiusa termina l'ambito iniziato con la
parentesi aperta più interna)

In ogni caso una variabile è visibile al programma e utilizzabile solo nello stesso ambito in cui è
dichiarata (variabili locali). Se si tenta di utilizzare una variabile in ambiti diversi da quello in cui è
dichiarata (o in ambiti superiori in caso di più ambiti innestati), il compilatore non la riconosce.

Il C++ ammette che si usi più volte lo stesso identificatore purché in ambiti diversi.

**Variabili globali**

Una variabile è globale, cioè è visibile in tutto il programma, solo se è dichiarata al di fuori di
qualunque ambito. Le dichiarazioni (con eventuali inizializzazioni) sono le uniche istruzioni del
linguaggio che possono anche risiedere esternamente all'ambito delle funzioni.

In caso di concorrenza fra una variabile globale e una locale viene riconosciuta la variabile locale.

**Variabili automatiche**

Una variabile è detta automatica (o dinamica), se cessa di esistere non appena il flusso del
programma esce dalla funzione in cui la variabile è definita. Se il flusso del programma torna nella
funzione, la variabile viene ricreata ex-novo e, in particolare, viene reinizializzata sempre con lo stesso
valore. Tutte le variabili locali sono, per default, automatiche ("tempo di vita" limitato all'esecuzione
della funzione).

**Variabili statiche**

Una variabile è detta statica se il suo "tempo di vita" coincide con l’intera durata del programma:

quando il flusso del programma torna in una funzione in cui è definita una variabile statica, ritrova la
variabile come l’aveva lasciata (cioè con lo stesso valore); ciò significa in particolare che l’istruzione di dichiarazione (con eventuale annessa inizializzazione) viene eseguita solo la prima volta.

Per ottenere che una variabile sia statica, bisogna preporre il qualificatore static nella dichiarazione
della variabile (davanti a tutti gli altri eventuali qualificatori). Esiste anche, per le variabili
automatiche, il qualificatore auto, ma è inutile in quanto di default (può essere usato per migliorare
la leggibilità del programma).

Una variabile statica, quindi, è locale ad una particolare funzione. E' inizializzata una sola volta, la prima volta che tale funzione viene chiamata e il suo valore resta inalterato quando si esce dalla funzione, per cui quando si richiama nuovamente la funzione tale variabile ha ancora il valore assegnatogli precedentemente.

Per definire statica una variabile è sufficiente anteporre la parola static alla dichiarazione della variabile.

Ad esempio:

 void stat(); /\* prototype function \*/

 int main(int argc, char\*\* argv){

int i;

 for(i=0; i<5; ++i)

 stat()

 }

 void stat(){

int auto\_var=0;

 static int static\_var=0;

 printf("auto=%d, static=%d \n", auto\_var, static\_var);

 ++auto\_var;

 ++static\_var;

 }

 Il cui output sarà:

 auto\_var=0, static\_var=0

 auto\_var=0, static\_var=1

 auto\_var=0, static\_var=2

 auto\_var=0, static\_var=3

 auto\_var=0, static\_var=4

La variabile auto\_var viene creata ogni volta, mentre la variabile static\_var è creata una sola volta ed il suo valore memorizzato.

**Esempio**

#include <iostream>

using namespace std;

int somma(int x,int y) {

 static int conta=0;

 conta++;

 cout<<conta;

 return(x+y);

}

int main(int argc, char\*\* argv){

 int a,b,c,s;

 cout<<"Dammi un numero: ";

 cin>>a;

 cout<<"Dammi un altro numero: ";

 cin>>b;

 cout<<"Dammi un altro numero: ";

 cin>>c;

 s = somma(a,b);

 s = somma(s,c);

 cout<<"La somma dei due numeri e' "<<c<<endl;

 return(0);

}

**Variabili globali**

Una variabile locale può essere automatica o statica; una variabile globale è sempre statica.

**APPROFONDIMENTO INVOCAZIONE DI UNA FUNZIONE**L’invocazione è il momento in cui il programmatore stabilisce di utilizzare la funzione. Questo momento comporta una deviazione del flusso di elaborazione e costringe il programma ad eseguire il copro della funzione.

In questa fase accadono i seguenti fatti:

1. Il codice del chiamante viene temporaneamente interrotto.
2. Gli argomenti indicati nella chiamata vengono passati ai parametri definiti con la funzione.
3. Viene eseguito il corpo della funzione.
4. Quando il codice del corpo termina OPPURE si incontra un comando return il controllo ritorna al chiamante.
5. Il codice del chiamante riceve il risultato della funzione e può proseguire.

x1, x2

**f(x1, x2)**

………

y = f(x)

………

return dato

**Main**

………

y = f(x1, x2)

………

Vediamo un semplice esempio:

#include <iostream>

using namespace std;

void ContoRovescia(int max){

for(int i=max; i>0; --i);

cout<<"Ora sono al numero: "<<i;

}

int main(int argc, char\*\* argv) {

ContoRovescia(5);

 return(0);

}

La funzione dell’esempio si chiama **ContoRovescia**, perché prende un numero e stampa il conteggio alla rovescia, senza restituire alcun valore e per questo si deve inserire un tipo nullo, il **void**.

Vediamo un altro esempio un po’ più complicato per calcolare la media dei valori contenuti in un array:

#include <iostream>

using namespace std;

void stampa(int valori[], int dim){

 for(int i=0;i<dim;i++){

 cout<<valori[i]<<" ";

 }

}

float media(int valori[], int dim){

 float somma=0;

 for(int i=0;i<dim;i++){

 somma+=valori[i];

 }

 return somma/dim;

}

int main(int argc, char\*\* argv){

 int valori[]={6, 5, 7, 8, 4, 10, 2, 6, 5, 7, 6, 7, 5};

 int dim=sizeof(valori)/sizeof(valori[0]);

 cout<<"Lunghezza del vettore: "<<dim<<endl;

 cout<<"La media dei seguenti valori: ";

 stampa(valori,dim);

 cout<<": "<<media(valori,dim);

 return(0);

}

In questo esempio abbiamo creato una funzione che calcola la media dei valori di un array di interi, il valore di ritorno non è intero perché la media di numeri interi è in virgola mobile il più delle volte.

L’esempio fornisce anche informazioni su come passare un array di valori ad una funzione.

**Esercizio**

Creare la funzione per restituire valori pari e dispari.

**Approfondimento Rispettare l’ordine nel passaggio dei parametri**

Quando si invoca una funzione contenente una lista di parametri allora nell’invocazione deve esserci lo stesso numero di argomenti, passati nello stesso ordine e con lo stesso tipo, ma senza **indicare il tipo degli argomenti**. Il tipo degli argomenti va specificato solo nella implementazione della funzione stessa. Quello che si deve sempre **indicare nell’invocazione è il modo del passaggio**. Inoltre se la funzione restituisce un tipo allora nell’invocazione si deve utilizzare lo stesso tipo.

Lista argomenti = <modo><tipo><nome>, <modo><tipo><nome> , . . . , <modo><tipo><nome>

**Esempio**//funzione senza parametri

//invocazione
string messaggio;
messaggio = msg() ; //funzione senza parametri

//definizione
public string msg(){
 return "ciao";
}

**Esempio**// funzione con 1 parametro

//invocazione
int y;
y=Quadrato(13); // funzione con 1 parametro

//definizione
public int Quadrato(int x){
 return x\*x;
}

**Esempio**// funzione senza tipo con 1 parametro

//invocazione
int y[];
Genera (y) ; // funzione senza tipo con 1 parametro

//definizione
static public void Genera(int y[]){
 for(int i=0;i<10;i++)
 y[i]=10–i;
}

**ESERCIZI DI BASE**

1. Dati tre numeri restituire la media dei due maggiori.
2. Dati tre numeri restituire quello “in mezzo”.
3. Definire una funzione che dice se un numero è perfetto; un numero è perfetto se è uguale alla somma dei suoi divisori (per esempio, 6 è un numero perfetto).
4. Definire una funzione che dice se un numero è primo; un numero è primo se è divisibile solo per sé stesso e per 1.
5. Definire una funzione che restituisce il fattoriale di un numero; il fattoriale di un numero è quel numero moltiplicato per i suoi predecessori.

**PASSAGGIO DEI PARAMETRI PER VALORE**

Quando si invoca una funzione occorre tenere ben distinti i parametri **formali** (quelli indicati nella definizione della funzione) dagli argomenti **attuali** (quelli specificati al momento della invocazione).

In particolare occorre tenere a mente che i parametri formali **esistono solo** nell’ambiente locale della funzione, e fuori da essa **non esistono**.

D’altra parte gli argomenti attuali della chiamata esistono solo se sono visibili nell’ambiente del chiamante, ovvero sono utilizzabili dal frammento di codice che sta invocando la funzione.

**Esempio**double Media(int x1, int x2){
 double y=(x1+x2)/2.0;
 return y;
}

int main(int argc, char\*\* argv){
 int a1=13;
 int a2=17;
 double y=Media(a1, a2);

 cout<<"La media e’: " <<y<<endl;

 system("pause");

 return 0;
}

Quando la funzione viene chiamata il sistema alloca memoria per poter eseguire la funzione. In questa memoria si sistemano spazi adatti sia ai parametri formali che alle variabili locali alla funzione.

Nell’esempio precedente i parametri (x1, x2) sono locali alla funzione che li usa come dei segnaposto per svolgere ipotetici calcoli.

Appena dopo aver creato questo spazio di memoria, il sistema provvede a copiare i valori degli argomenti attuali nelle locazioni dei parametri formali. Si faccia attenzione: le locazioni sono ben distinte anche se i nomi possono coincidere. Quello che avviene è che le locazioni associate ai parametri formali vengono riempite da copie dei valori contenuti negli argomenti attuali.
Nell’esempio precedente quando si esegue l’invocazione con gli argomenti (a1, a2) i loro valori vengono copiati negli argomenti (x1, x2). Una volta acquisiti i valori la funzione può svolgere i calcoli e, eventualmente, restituire un risultato (nel nostro caso y).

Nel passaggio di parametri per valore non si specifica alcun modo nel parametro. I valori degli argomenti sono copiati nei parametri. I parametri lavorano su copie dei valori, non sugli originali.

**PASSAGGIO DEI PARAMETRI PER RIFERIMENTO**

Abbiamo appena visto che nel passaggio per valore i parametri lavorano su copie dei valori degli argomenti. Per questo eventuali modifiche sui parametri **non hanno effetto** sugli argomenti. Questo effetto può garantire sicurezza in alcuni casi ma può avere effetti indesiderati. Si consideri, infatti, il seguente esempio:

**Esempio**

|  |  |
| --- | --- |
| **Num** | **Codice** |
| 12345678910111213141516 | #include <iostream>using namespace std;void scambia(int x1, int x2){ int tmp=x1; x1=x2; x2=tmp;}int main(int argc, char\*\* argv){  int a1=14, a2 = 23; cout<<"a1 = "<<a1<<" ; "<<"a2 = "<<a2<<endl; scambia(a1, a2); cout<<"a1 = "<<a1<<" ; "<<"a2 = "<<a2<<endl; return(0);} |

Nello studiare una funzione occorre ricordarsi che “quello che vorrei che facesse la funzione” non sempre coincide con “quello che effettivamente fa la funzione”.

Nell’esempio appena proposto si vorrebbe scambiare il contenuto delle due variabili a1 e a2. Verifichiamo se questo corrisponde a quello che la funzione fa effettivamente utilizzando la tabella di traccia:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Num** | **a1** | **a2** | **x1** | **x2** | **tmp** | **OUT** |
| 11 |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |
| 13 | 14 | 23 |  |  |  |  |
| 14 |  |  |  |  |  | a1 = 14 ; a2 = 23 |
| 15 |  |  |  |  |  |  |
| 4 |  |  | 14 | 23 |  |  |
| 12 |  |  |  |  |  |  |
| 6 |  |  |  |  | 14 |  |
| 7 |  |  | 23 |  |  |  |
| 8 |  |  |  | 14 |  |  |
| 9 |  |  |  |  |  |  |
| 16 |  |  |  |  |  | a1 = 14; a2 = 23 |
| 17 |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |

Quando la funzione viene invocata il sistema alloca memoria per la funzione, per i suoi parametri (x1 ed x2) e per le variabili locali (tmp).

Il passaggio dei parametri consente di riempire le locazioni dei parametri formali coi valori 14 e 34.

Dopo aver ricevuto i valori viene eseguito il codice della funzione che scambia il contenuto di x1 con quello di x2. Adesso il controllo torna al chiamante che prosegue, ma… quando confronta a1 con a2 **evidentemente** i loro valori **non** sono cambiati, poiché gli unici a cambiare sono stati x1 ed x2.

Quando si desidera che gli argomenti passati possano essere modificati dalla funzione, occorre passarli per riferimento. Il passaggio per riferimento permette alla funzione di agire sugli argomenti che il programma chiamante ha passato.

In C++ il passaggio di un parametro per riferimento si realizza anteponendo al nome della variabile il simbolo di e-commerciale (&).

L’esempio precedente riveduto diventa il seguente:

#include <iostream>

using namespace std;

void scambia(int &x1, int &x2){

 int tmp=x1;

 x1=x2;

 x2=tmp;

}

int main(int argc, char\*\* argv){

 int a1=14, a2 = 23;

 cout<<"a1 = "<<a1<<"\t"<<"a2 = "<<a2<<endl;

 scambia(a1, a2);

 cout<<"a1 = "<<a1<<"\t"<<"a2 = "<<a2<<endl;

 return(0);

}

Nell’esempio precedente gli argomenti (a1, a2) vengono passati per **riferimento**; quando viene invocata la funzione in realtà **non** c’è alcuna copia di valori, ma si condividono le stesse locazioni di memoria. In pratica ogni volta che la funzione lavora sui parametri (x1, x2) effettivamente il lavoro viene effettuato sugli argomenti (a1, a2). Si osservi che in ogni chiamata detti argomenti cambiano.
Per esempio quando x1 cambia valore nell’esempio, la locazione che modifica realmente il valore sarà l’argomento a1. La stessa cosa accade per x2 con a2.

**APPROFONDIMENTO PASSAGGIO DI PARAMETRI PER INDIRIZZO**

Prima di affrontare il passaggio per riferimento, ci aiuterà esaminare un altro modo per passare gli argomenti ad una funzione: quello del passaggio per indirizzo, molto simile a quello per riferimento, in cui è necessario passare alla funzione degli argomenti di tipo puntatore.

Un vantaggio in più del passaggio per indirizzo rispetto a quello per valore è che la variabile può restituire più di un valore. Facciamo subito un esempio in cui riproponiamo la funzione che calcola la moltiplicazione tra due numeri ma con i parametri passati per indirizzo.

#include <iostream.h>

using namespace std;

// Definizione del prototipo

int moltiplica(int\* x, int\* y);

main(){

 int a = 5;

 int b = 6;

 int risultato;

 risultato = moltiplica(&a, &b);

 cout << Il risultato della moltiplicazione è: << risultato <<endl;

 return (0);

}

// Dichiarazione della funzione

int moltiplica(int\* x, int\* y){

 int ris;

 // effettua la moltiplicazione tra i valori che si

 // trovano leggendo il contenuto della memoria

 // agli indirizzi x e y, tramite il prefisso '\*'

 // (da non confondere con l'operatore di moltiplicazione)

 ris = \*x \* \*y;

 // Valore restituito dalla funzione.

 return ris;

}

**Analisi di una funzione**

Vediamo quali sono le domande tipiche che ci dobbiamo porre nell’utilizzare le funzioni:

1. La funzione restituisce un valore?
	1. Se SI, allora occorre specificare il tipo (il risultato è SEMPRE UNO SOLO).
	2. Se NO, allora il tipo è VOID (che significa nessun tipo).
2. La funzione può rispondere al suo compito senza informazioni dal chiamante?
	1. Se SI, allora la funzione è SENZA PARAMETRI.
	2. Se NO, allora occorre individuare con certezza i PARAMETRI NECESSARI.
3. La funzione può dover restituire in una locazione delle informazioni in uscita?
	1. Se SI, allora occorre marcare quei parametri come PASSATI PER RISULTATO.
	2. Se NO, allora vedi la domanda 4.
4. La funzione può dover modificare una delle informazioni in ingresso?
	1. Se SI, allora occorre marcare quei parametri come PASSATI PER RIFERIMENTO.
	2. Se NO, allora occorre lasciare quei parametri PASSATI PER VALORE.
5. Che nome ha la funzione?

**ESERCIZI**

1. Definire una funzione che presi due numeri renda la loro media.
2. Definire una funzione che presi tre numeri renda sia il minimo che il massimo.
3. Definire una funzione che allochi memoria e ponga numeri casuali in un vettore di double.
4. Definire una funzione che restituisca il minimo valore in un vettore di double.
5. Definire una funzione che preso un vettore di interi restituisca un nuovo vettore con i soli valori pari contenuti nel primo vettore.
6. Definire una funzione che preso un vettore di interi scambi di posto il valore minimo col massimo.
7. Definire una funzione che presi due vettori di interi controlli se sono uguali.
8. Definire una funzione che presi due vettori di interi restituisca un nuovo vettore che contenga i valori di entrambi i vettori.
9. Definire una funzione che presi due vettori di interi già ordinati restituisca un nuovo vettore che contenga i valori di entrambi i vettori già ordinati.
10. Definire una funzione che preso un vettore di interi lo ordini in modo crescente.
11. Definire una funzione che restituisca il minimo valore in una matrice bidimensionale di double.
12. Definire una funzione che restituisca le coordinate del minimo valore in una matrice bidimensionale di double.
13. Definire una funzione che prese due matrici di interi controlli se sono uguali.
14. Dichiarare la funzione che genera un vettore di interi casuali ma di dimensione specificata per parametro.
15. Dichiarare una funzione che controlla se due vettori di interi sono uguali (stessi valori nelle stesse posizioni); invocarla da pulsante.
16. Dichiarare una funzione che cerca in un vettore di interi un determinato valore; se lo trova restituisce la prima posizione in cui lo ha trovato; altrimenti rende -1.
17. Dichiarare una funzione che controlla se due vettori hanno almeno un elemento in comune.
18. Dichiarare una funzione che scambia di posizione il valore minimo e massimo in un vettore; in caso di valori uguali sceglierne uno qualsiasi.
19. Dichiarare una funzione che restituisce una copia di un vettore ma composto dei soli numeri pari; ovviamente il vettore copia è più piccolo dell’originale (al massimo ha stessa dimensione).
20. Dichiarare una funzione che preso un vettore di interi e due estremi, crea e restituisce un secondo vettore copiando gli elementi compresi nelle posizioni specificate.
21. Dichiarare una funzione che prende due vettori e rende una matrice di dimensioni adeguate tale che ogni cella di coordinate [ r , c ] ha come valore il prodotto della cella [r] del primo vettore per la cella [c] del secondo.
22. Dichiarare una funzione che prende una matrice quadrata di ordine N (ha N righe e N colonne!!!) e la traspone (ovvero scambia di posizione gli elementi rispetto alla diagonale principale).
23. Dichiarare una funzione che presa una matrice quadrata di ordine N, una dimensione e le coordinate di una cella, restituisce la matrice (detta minore) che parte da quella cella ed ha quella dimensione; la dimensione deve essere minore di N.
1. Probabilmente il doppio simbolo ++ contenuto nel termine C++ deriva dall’istruzione d’incremento unitario. Per esempio se scriviamo conta++; equivale a scrivere conta = conta + 1; [↑](#footnote-ref-1)
2. La At & T (American Telephone and Telegraph Incorporated) è una delle più importanti compagnie telefoniche degli stati uniti. [↑](#footnote-ref-2)
3. Fate attenzione a come scrivere i numeri con la virgola; la nostra virgola deve essere scritta con il punto (sistema anglosassose) [↑](#footnote-ref-3)