Indice programmazione 1:

Lezione 1 p.1: Da problemi ad algoritmi

Lezione 2 p.2: C-C++, variabili, int

Lezione 3 p.6: assegnamento sintassi, bool, espressioni

Lezione 4 p.8: programmazione strutturata, istruzioni scelta

Lezione 5 p.10: istruzioni cicliche

Lezione 6 p.13: notazione posizionale

Lezione 7 p.15: funzioni

Lezione 8 p.19: visibilità, tempo di vita

Lezione 9 p.21: Comportamento bool, char, conversioni esplicite

Lezione 10 p.22: Enum, reali, conversione tipo

Lezione 11 p.26: Passaggio per riferimento

Lezione 12 p.27: Ingegneria del codice

Lezione 13 p.28: Array

Lezione 14 p.30: Stream file, Input/output formattato

Lezione 15 p.34: Stringhe, struct, matrici

Lezione 16 p.36: Array dinamici, puntatori

Lezione 17 p.39: Input/output non formattato

Lezione 18 p.41: Compendio C/C++

Lezione 19 p.43: Algoritmi, linguaggio, traduzione, comp.

Lezione 20 p.47: Gestione memoria

Lezione 21 p.50: Introduzione Liste

# Programmazione 1

## LEZIONE 1

Per poter parlare di algoritmo è necessario che le azioni da compiere siano :

- Perfettamente definite
- Assolutamente non ambigue

Gli algoritmi che faremo produrranno **informazioni**.

Computer: Macchina in grado di eseguire insiemi di azioni elementari -> le azioni vengono eseguite su oggetti in ingresso, per produrre oggetti in uscita -> all'elaboratore vengono richieste azioni tramite frasi scritte in un linguaggio **(istruzioni).**

Per dire al calcolatore cosa fare serve il **linguaggio di programmazione**.

Il linguaggio dispone di propria **sintassi** (simboli, parole ammesse, regole grammaticali)

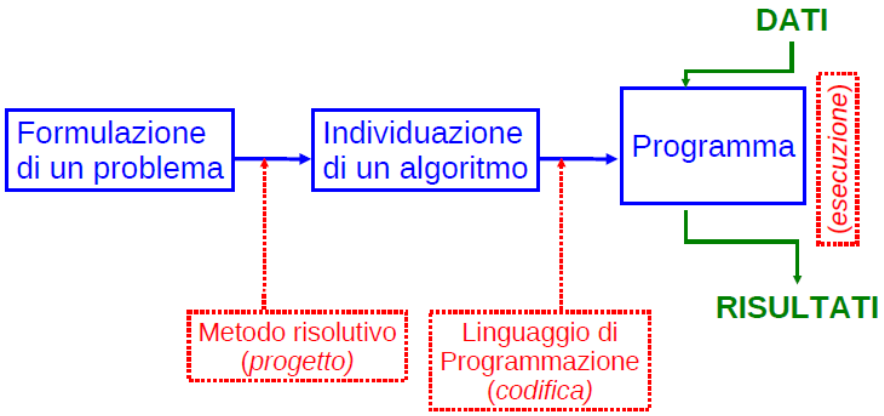
e **semantica** significato dei simboli e delle parole, mentre alcune parole sono istruzioni (azioni da compiere)

Programma= documento di testo scritto con sintassi e semantica del LDP

Programmazione, codifica o implementazione= Scrittura algoritmo seguendo un insieme di frasi appartenenti al linguaggio di programmazione che specificano le azioni da compiere (in pratica scrittura di un programma)

In sostanza**: un programma è la formulazione testuale di un algoritmo in un LDP**

Sintassi e semantica definite in modo rigido e completo, l'esecuzione delle azioni avviene nell'ordine specificato dall'algoritmo per ottenere i risultati che risolvono il problema




## ESERCITAZIONE: INGRESSO USCITA

Un programma C/C++ è un documento di testo scritto in linguaggio C/C++

I suffissi tipici dei file contenenti programmi sono per C -> .c per C++->.cc

Input= Ingresso di informazioni (da elaborare) all'interno di un processo

Output= Uscita di informazioni (elaborate) da un processo

Un programma che vuole effettuare un I/O classico deve contenere le direttive (precedono il primo punto in cui viene effettuato I/O):

#include <iostream>

Using namespace std;

Sintassi per stampare una stringa "cout<<stringa", ove stringa è una sequenza di caratteri delimitata da doppi apici ".

I microprocessori degli elaboratori ragionano in **linguaggio macchina,** un linguaggio elementare detto di basso livello (non facile per scrivere programmi, si usano infatti linguaggi di alto livello).

Per poter essere eseguito, ossia diventare un "Eseguibile" un programma deve essere tradotto in linguaggio macchina, questa operazione è effettuata da strumenti chiamati **compilatori**.

C/C++ è un ling. Di alto livello, bisogna quindi tradurlo prima di poterlo usare.

File sorgente = file di testo che contiene programma scritto nel linguaggio di partenza

**SORGENTE -> COMPILAZIONE -> ESEGUIBILE**

Sintassi più semplice per generare un eseguibile da file sorgente è: g++ nome_sorgente.cc

Assegna un nome predefinito al programma eseguibile (./a.out)

g++ -o nome_eseguibile nome_sorgente.cc permette di scegliere il nome del programma eseguibile.

Se ci sono problemi il compilatore può comunicare:

- Warning (avvisi): c'è qualcosa di "sospetto" nel codice, ma si può comunque compilare
- Error: ci sono errori che impediscono la compilazione

Per invocare un programma nella cartella corrente abbiamo due possibilità:

1) Percorso assoluto ES:/home/giovanni/a.out
2) Percorso relativo dicendo alla shell di cercare il file **qui**, per farlo usiamo il nome speciale "." ES: ./a.out

Caratteri speciali:

- \n newline
- \t tabulazione
- \\ barra inversa
- \' apice
- \" virgolette

Gli operatori di uscita << possono essere accordati l'uno all'altro, e non solo stringe posso essere passate come argomento, ES: endl = \n.

LEZIONE 2

In C/C++ si possono scrivere numeri interi (6, 12, 700), e si possono inoltre definire delle variabili di tipo **int**

Variabili = contenitori in cui memorizzare numeri interi (un sottoinsieme limitato di numeri interi), il contenuto può cambiare durante l'esecuzione del programma

Per poter usare una variabile bisogna prima definirla, stabilendone: **l'identificatore** (il nome che useremo per riferirci alla variabile), e opzionalmente possiamo **inizializzarla**.

**Processo=** programma in esecuzione, comando top per vedere stato dei processi su Linux.

Processo I/O realizzato facendo entrare o uscire dal programma flussi di caratteri

Flusso di caratteri = successione di righe, ciascuna costituita da zero o più caratteri e terminata dal carattere speciale '\n'.

C++ non prevede istruzioni per ingresso o uscita, ma vengono implementati tramite *oggetti di libreria* chiamati *stream*

Stream= meccanismo generale per far entrare o uscire dal programma flussi di caratteri

Ostream = meccanismo per gestire flusso di caratteri in uscita, output formattato

Istream= Meccanismo Per gestire flusso di caratteri in ingresso, input formattato.

Appena un programma inizia ci sono già tre flussi aperti:

cin: flusso standard d'ingresso (lettura dei caratteri immessi dal terminale in cui è in esecuzione la shell)

cout: flusso standard d'uscita (Visualizz.ne sul terminale di cui è in esecuzione la shell)

cerr: flusso standard di uscita per comunicare messaggio d'errore (Visualizz.ne sul terminale di cui è in esecuzione la shell)

Operatore di uscita <<, può stampare una stringa o un manipolatore.

Un processo ritorna un valore quando termina, 0 tipicamente indica che è andato tutto bene; la funzione main deve ritornare un valore di tipo intero, ed è il valore ritornato dal processo stesso, se non si dichiara di tipo int e/o non si fa ritornare un valore con return può dare errore o warning

Operatore >> applicato ad un oggetto di tipo istream legge i caratteri in ingresso dallo stdin e li interpreta in base al tipo della variabile

Se il valore immesso non è in notazione decimale, la lettura fallisce e il cin entra in stato di errore facendo fallire le successive letture.

Variabile = contenitore in cui si può memorizzare un valore, che può variare nel tempo

In c++ bisogna elencare ogni variabile prima che venga utilizzata, (=**definizione)** e bisogna attribuirle un **tipo ,** un **identificatore** ed eventualmente **inizializzarla.**

Visibilità variabile= una var è visibile solo dal punto in cui viene dichiarata.

Sviluppo soluzione di un algoritmo:

1) Riflettere sul problema finchè non si è sicuro di aver capito a sufficienza tutti gli aspetti e implicazioni
2) Cercare di farsi venire l'idea che sembri buona per risolvere il problema (o in parte)
3) Provare a definire l'algoritmo e verificare per capire se è corretto
4) Se si è sicuri dell'algoritmo, partire con la codifica
5) Collaudare il programma

**Memoria (**principale) = il contenitore in cui sono memorizzati tutti i dati su cui lavora il processore

Possiamo schematizzare la memoria come una sequenza contigua di celle (anche dette locazioni di memoria)

Ciascuna cella è l'unità minima di memorizzazione, ossia l'elemento più piccolo in cui memorizzare un'informazione

Ogni cella contiene un byte, ossia una sequenza di bit; un byte è costituito da 8 bit.

Tutte le celle hanno quindi la stessa dimensione in termini di numero di bit, mentre il numero di bit in un byte potrebbe variare da macchina a macchina

Ciascuna cella è **univocamente** individuata mediante un numero naturale chiamato indirizzo della cella

I bit contenuti in una cella possono essere utilizzati per memorizzare un numero tramite notazione binaria

Per memorizzare numeri maggiori di 255(11111111) si **accorpano** più celle consecutive (Una variabile di tipo int è tipicamente rappresentata su 4 celle consecutive)

PROCESSORE= tutte le operazioni di elaborazione delle info. Effettuate da un calcolatore sono svolte direttamente dal processore, oppure da altre componenti dietro comando del processore.

Il proc. Può compiere operazioni molto semplici (lettura, scrittura, somma, sottr, ecc) e può lavorare su un certo numero di celle contigue alla volta, tale sequenza è detta (**parola di macchina**)

Ogni processore è caratterizzato da un proprio insieme di **istruzioni** (per fargli svolgere le precedenti operazioni)

L'insieme delle istruzioni viene chiamato **linguaggio macchina**, ogni istruzione è identificata da una configurazione di bit.

Per far eseguire un programma ad un processore basta: memorizzare da qualche parte nella memoria la sequenza di configurazioni di bit relativa alle istruzioni da eseguire, e dire al processore in quale indirizzo si trova la prima di tali istruzioni, dopo di ché, il processore eseguirà una dopo l'altra le istruzioni che trova a partire da quella indicata come iniziale.

L'ordine cambia solo se vengono incontrate *istruzioni di salto* verso un altro indirizzo (può andare avanti o indietro di indirizzo, es: cicli)

Data la difficoltà di programmare in linguaggio macchina, sono stati inventanti linguaggi cosiddetti di *alto livello*, che si basano sul concetto di **astrazione** : astraggono dei dettagli cosiddetti di basso livello (come celle di mem, e indirizzi) permettendo di scrivere il programma in termini di dati e operazioni più complessi

**Memoria=** (o *processo*) il contenitore logico in cui sono memorizzati tutti i dati del programma (ed altro) durante la sua esecuzione

Tale memoria ha la stessa struttura di quella del calcolatore, una sequenza contigua di **celle** (locazioni di memoria) che costituiscono l'unità minima di memorizzazione, le celle sono tutte di ugual dimensione, ovvero un byte

(l'esatta dimensione di un byte non è specificata nello standard di linguaggio, ma **deve** essere abbastanza grande da contenere un char)

In C/C++ si possono memorizzare informazioni più complesse con una singola cella di memoria, all'interno di contenitori genericamente chiamati oggetti

**Oggetto=** astrazione di una cella di memoria, caratterizzato da un **valore**, e memorizzato in una **sequenza di celle contigue**

In un oggetto troveremo:

- Un indirizzo ( es: 232, l'oggetto si troverà in memoria a partire dalla cella di indirizzo 232)
- Un valore ( il contenuto della cella/e)
- Un **tipo (di dato** )= i valori possibili per l'oggetto e le operazioni che si possono effettuare sull'ogg.

## IL TIPO **INT**

È diverso dal tipo intero in senso matematico, ovvero, il tipo **int** ha un insieme di valori limitato, l'insieme esatto dei valori possibili dipende dalla macchina e dal sistema installato su di essa( normalmente configurati con al più lo spazio di una PAROLA DI MACCHINA [2,4,8 BYTE ossia 16,32,64 bit], es se la macchina ha parola a 32 bit: [-2^31, 2^31-1], ovvero [-2147483648, +2147483647]).

Sono utilizzabili gli operatori: + - * / %

ESPRESSIONI LETTERALI

Le espressioni letterali denotano valori costanti, spesso chiamate semplicemente letterali o costanti senza nome

COSTANTI CON NOME= Associa permanentemente un oggetto di valore costante ad un identificatore

La definizione è identica a quella di una variabile, a parte l'aggiunta della parola chiave **const** all'inizio + l'obbligo di Inizializzazione.

Costante= astrazione **simbolica** di un valore, si da cioè un nome ad un valore, è un'associazione identificatore-valore che non cambia mai durante l'esecuzione -> non si può effettuare un'operazione di assegnamento.

Nel caso di una variabile l'associazione identificatore- indirizzo non cambia mai durante l'esecuzione, ma può cambiare l'associazione identificatore-valore

STRUTTURA SEMPLIFICATA DI UN PROGRAMMA

In prog. 1 solo su un unico file

Pre-processore= prima della compilazione, il file sorg. Viene manipolato dal suddetto, ha il compito di effettuare delle modifiche o aggiunte al testo originario, (viene pilotato dalle **direttive**)

La nuova versione del file viene quindi memorizzata in un file temporaneo, ed è questo file che passa al compilatore (file temporaneo automaticamente distrutto alla fine della compilazione)

**Dichiarazione=** è un'istruzione in cui si introduce un nuovo identificatore

Le definizioni sono in caso particolare di dichiarazione: sono dic. la cui esecuzione alloca spazio in memoria.

(es : #include <iostream> → è una direttiva per il Pre-Processore)

Main() è una funz. Speciale con 3 caratteristiche:

1) Deve essere sempre presente
2) La prima operazione svolta dal calcolatore è la prima istruzione del main, ovunque esso sia
3) Dopo l'ultima istruzione del main, termina il programma intero

Le istruzioni vengono eseguite in sequenza o concatenazione (= una sequenza di istruzioni scritte di seguito una all'altra all'interno del programma), le istruzioni vengono eseguite una dopo l'altra.

## Lezione 3

Espressione di assegnamento: Nome_variabile= <espressione>

Istruzione di assegnamento= espressione di assegnamento **seguita da un ;** (viene utilizzata per assegnare un valore a solo variabili)

Nell'assegnamento viene preso l'indirizzo della variabile individuata dall'identificatore a sinistra dell'assegnamento (tale indirizzo è detto **lvalue (left value)**

Viene calcolato il valore dell'espressione che compare a destra ed assegnata all'oggetto memorizzato all'indirizzo lvalue, tale valore è detto **rvalue (right value)**

L'assegnamento comporta prima la valutazione di tutta l'espressione a destra dell'assegnamento.

Solo **dopo** si inserisce il valore risultante (rvalue)

Il valore dell'espressione di assegnamento è *l'indirizzo della variabile a cui si è assegnato il nuovo valore* (l'lvalue) es: l'espress. A=3 ha per valore l'indirizzo di a

Sintassi del C/C++

Un programma C/C++ è una sequenza di parole (**token**) delimitate da spazi bianchi (**whitespace)**

Spazio bianco= carattere spazio, tab, a capo

Parola = sequenza di lettere o cifre non separate da spazi bianchi (token possibili: operatori, separatori, identificati, parole chiave(riservate), commenti, espressioni letterali)

IDENTIFICATORI: lettera-lettera|cifra

Lettera = tutte le lettere, maiuscole, minuscole e l'undescore(_)

Lettera|cifra= sequenza indefinita di elementi Lettera o Cifra,

Maiuscole o minuscole sono considerate **diverse** (il linguaggio C/C++ è *case-sensitive*)

PAROLE CHIAVE (RISERVATE): Int, float, double ecc..

COMMENTI: // Commento, su una sola riga

/*commento anche su più righe*/

Parola chiave e identificatore vanno separati da almeno uno spazio bianco, in tutti gli alti casi, gli spazi bianchi non sono obbligatori (li si usa per dare leggibilità) e si può separare token consecutivi con il numero e il tipo di spazi bianchi che si preferisce

TIPO BOOLEANO (bool)

Valori possibili: vero (true) e falso (false)

Operazioni possibili: Not, and, or

Se non inizializzata una variabile assume valore casuale

Se un oggetto di tipo booleano viene usato dove è atteso un val numerico, True viene convertito a 1 e False a 0 ; viceversa se un oggetto numerabile è utilizzato dove è atteso un booleano, ogni valore diverso da 0 è convertito a true, il valore 0 a false.

OPERATORI DI CONFRONTO:

== operatore di confronto di uguaglianza ( solo il simbolo = denota l'op di assegnamento)

!= operatore di confronto di diversità

> op di confronto di maggiore stretto

< di minore stretto

>= maggiore uguale

<= minore uguale

RESTITUISCONO TUTTI UN TIPO **BOOLEANO**

Assegnamento abbreviato: **a += b <-> a = a + b**

Incremento e decremento: ++ --

Prefisso: prima si effettua l'incremento/decremento poi si usa la variabile (restituisce lvalue)

Postfisso: prima si usa il valore della variabile, poi si effettua l'incr/decr. (restituisce rvalue)

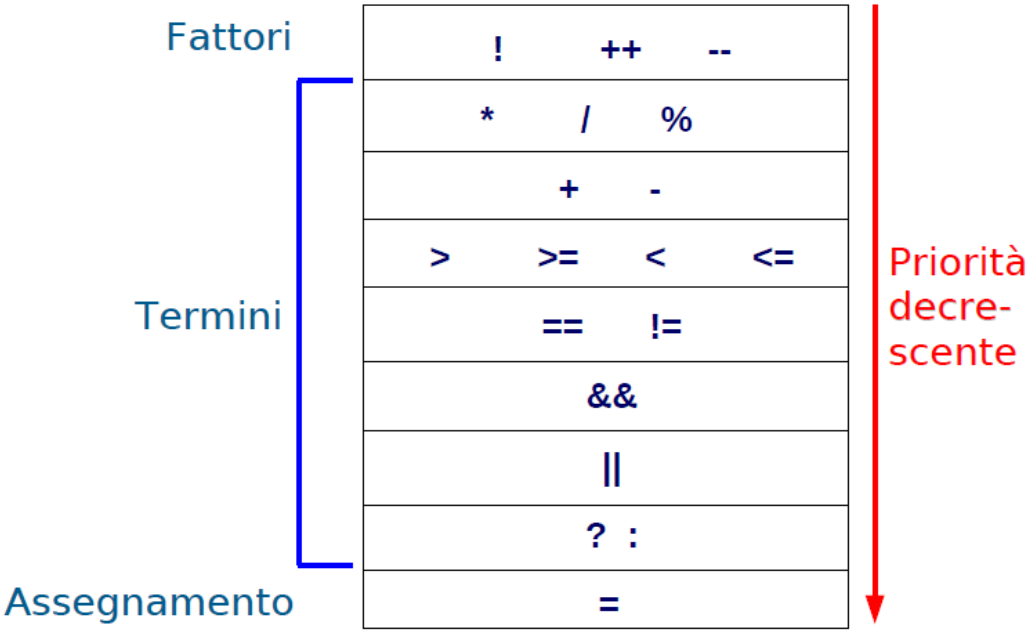
Un'espressione di definisce **aritmetica** se produce un risultato aritmetico o **logica** se produce un risultato booleano

PRIORITÀ DEGLI OPERATORI

- Posizione rispetto ai suoi operandi (o argomenti) : prefisso, postfisso, infisso
- Numero di operandi : unari, binari e ternari
- Precedenza (o priorità) nell'ordine di esecuzione
- Associatività, due operatori con la stessa precedenza possono essere associativi a sinistra (es 6/3/2 è uguale a (6/3)/2) oppure associativi a destra (es: = e associativo a destra)
  Infatti l'associatività dell'operatore di assegnamento è a destra

Ordine di valutazione delle espressioni: prima i fattori (ottenuti dalle espressioni letterali e dal calcolo delle funzioni e degli operatori unari) poi i termini (ottenuti dal calcolo degli operatori binari)

Sintesi priorità degli operatori:




Quando si progetta un programma, prima carta e penna, poi ci si mette nei panni del compilatore e dell'esecutore

Lezione 4

Salto= è la prima tecnica adottata storicamente per eseguire passi diversi a seconda dei dati passati in ingresso (goto, nel linguaggio d'alto livello, jump nel linguaggio macchina)

Con un salto all'indietro è possibile ripetere pezzi di codice in maniera ciclica

Non studieremo i salti per evitare la logica a spaghetti

**Programmazione strutturata:** se si utilizzano solo i seguenti costrutti per determinare l'ordine di esecuzione delle istruzioni (anche detto flusso di controllo):

- **Concatenazione e composizione :** vedi concatenazione, mentre composizione permette di trattare una sequenza di istruzioni come se fosse una sola istruzione
- **Selezione (istruzione condizionale):** fa seguire il flusso di controllo tra due possibili rami in base al valore vero o falso di una espressione detta condizione di scelta
- **Iterazione** : Permette l'esecuzione di un'istruzione o di una sequenza finché rimane vera l'espressione detta "Condizione di iterazione"

Scopo è quello di rendere più leggibili e facile i programmi, ma rischiamo di non essere in grado di codificare qualche algoritmo?

MACCHINA DI TURING , dotata di una testina e un nastro costituito da celle adiacenti (ipoteticamente) infinite

Ogni algoritmo può essere eseguito (calcolato) dalla macchina di Turing (questa tesi è indimostrabile o perlomeno mai dimostrata, ma è universalmente accettata

Tesi di Church-Turing = ogni algoritmo può essere tradotto in un programma scritto caratterizzato solo da: tipo di dato → naturali con l'operazione (+) e istruzioni → assegnamento, istr. Composta, condizionale o di iterazione,

quindi la programmazione strutturata può esprimere qualsiasi argomento

ISTRUZIONI CONDIZIONALI:

due tipi: 1) istruzione semplice o alternativa e 2) istruzione multipla (migliora la leggibilità)

SCELTA SEMPLICE : consente di scegliere fra due istruzioni alternative in base al verificarsi o meno di una data condizione: If (<condizione)

<istruzione1>

[else

<istruzione 2>]

<condizione> è un'espressione logica valutata al momento di esecuzione dell'istruzione if

Se <condizione> risulta vera, si esegue <istruzione 1> altrimenti <istruzione2>; in entrambi i casi l'esecuzione continua e va dopo l'if

ISTRUZIONE COMPOSTA: sequenza di istruzioni racchiuse tra parentesi graffe {<istruzione 1> <istruzione2>…}; ai fini della semantica e della sintassi, una istruzione composta è trattata come una singola istruzione semplice, l'esecuzione di una istruzione composta implica l'esecuzione ordinata di tutte le istruzioni della sequenza tra le graffe

Entrambe le istruzioni, del ramo if ed else possono essere una qualsiasi istruzione semplice o composta.

Nel caso di un'operazione illegale il processore genera una eccezione hardware, all'avvio iniziale, il sistema operativo associa ad ogni eccezione hardware del codice di gestione dell'eccezione stessa (nel caso di una divisione per zero, termina forzatamente il processo che ha generato l'eccezione)

La terminazione forzata è una delle modalità di fallimento di un programma, come la restituzione di risultati errati.

Indentazione: tutte le istruzioni appartenenti al loro corpo devono essere allineate a partire da una colonna

ISTRUZIONI DI SCELTA ANNIDATE If all'interno di condizioni degli If

```
If (n>0)
        If (a>b) n=a;
        else n=b*5;
```

l'else è sempre associato all'if più interno (vicino); se questa non è l'associazione desiderata racchiudere l'if più interno in un blocco {}.

Istruzione di scelta multipla: costrutto sintattico che ci permetta di eseguire solo l'istruzione giusta in base al valore della variabile; consente di scegliere fra molti casi in base al valore di un'**espressione di selezione**

**switch (***<espressione di selezione>***) {**
    **case** *<etichetta1>* **:** *<sequenza_istruzioni1>* [ **break;**]
    **case** *<etichetta2>* **:** *<sequenza_istruzioni2>* [ **break;** ]

…
[ **default :** < *sequenza_istruzioniN*> ]
}

<espressione di selezione> restituisce un valore **numerabile** e viene valutata al momento dell'esecuzione dell'istruzione switch, le ***etichette*** devono essere delle costanti dello stesso tipo dell'espressione selezione

**Corpo dello switch =** parte compresa tra le parentesi graffe

Il valore dell'espressione di selezione viene confrontato con le costanti che etichettano i vari casi e l'esecuzione salta al ramo dell'etichetta corrispondente (se esiste)

Dopo il salto al ramo di una delle etichette prosegue sequenzialmente fino alla fine del corpo dell'istruzione switch, a meno che non incontri un'istruzione break, in quel caso si esce dal corpo dello switch e l'esecuzione prosegue dall'istruzione successiva all'istruzione switch.

Se nessuna etichetta corrisponde al valore dell'espressione, si salta al ramo default (se specificato)

I rami non sono mutuamente esclusivi, una volta saltato all'inizio di un ramo, l'esecuzione prosegue con le istruzioni dei rami successivi fino alla fine del corpo switch; per avere rami mutuamente esclusivi, occorre forzare esplicitamente l'uscita mediante l'istruzione break

Pro e contro: lo switch garantisce più leggibilità, ma è utilizzabile solo con espressioni ed etichette di tipo numerabile (intero, carattere, enumerato), e non è utilizzabile con num. Reali o strutturati.

## ESERCITAZIONE 4

Per descrivere il comportamento di un programma o quello che accade nel suo sviluppo, si fa riferimento alle azioni che possono essere compiute da questi 4 attori: **programmatore, compilatore, il programma stesso e l'utente.**

Tre diversi momenti : a tempo di scrittura da parte del programmatore, a tempo di compilazione da parte del compilatore e a tempo di esecuzione del programma da parte dell'utente o del programma stesso

Operatore sizeof: restituisce la dimensione di un'espressione o di un tipo

Sizeof (espressione) = numero di byte (char) necessari per memorizzare i possibili valori dell'espressione

Sizeof (nome_tipo) = numero di byte (char) necessari per memorizzare un oggetto del tipo passato come parametro

Overflow= si ha quando il valore di un'espressione è troppo grande (in modulo) per essere contenuto nel tipo di dato del risultato o nell'oggetto a cui si vuole assegnare tale valore → in tal caso il risultato sarà in generale **non correlato** con l'operazione effettuata.

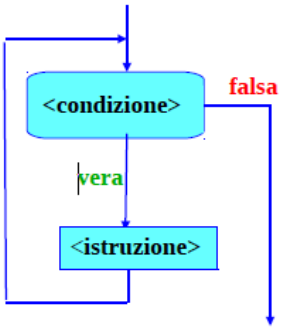
Lo standard non prescrive segnalazioni di errore di overflow a tempo di esecuzione, quindi le operazioni sono effettuate senza controllare se il risultato sarà corretto.

## LEZIONE 5

Ultimo costrutto fondamentale della programmazione strutturata: **istruzioni iterative**

Le istr. Iterative (o cicliche) Forniscono costrutti di controllo che **permettono di ripetere una certa istruzione fintanto che una certa condizione è vera;** per migliorare l'espressività del linguaggio, il C/C++ fornisce vari tipi di istruzioni iterative (cicliche) : while, do….while, for.

L'istruzione da ripetere finché la condizione rimane vera, viene chiamata **corpo del ciclo**, ogni ripetizione dell'esecuzione del corpo viene chiamata **iterazione (del ciclo)**




WHILE

While (<condizione>) <istruzione>

Istruzione: è il corpo del ciclo e viene ripetuta per tutto il tempo in cui <condizione> rimane vera

Se <condizione > è già inizialmente falsa, il ciclo non viene eseguito neppure una volta; infatti non è noto a priori se e quante volte l'istruzione verrà eseguita.

L' istruzione deve modificare la condizione altrimenti si ha un ciclo infinito, infatti molto spesso è un'istruzione composta che tra le varie istruzioni modifica qualcuna delle variabili che compongono la condizione

Comando della shell: sleep <numero_secondi> aspetta il numero di secondi e poi termina

System ("riga_di_comando") : invoca i comandi da dentro un programma C/C++; il programma rimane fermo finchè il comando non è terminato (Per usare la system, bisogna aggiungere la direttiva #include <stdlib.h>)

Dopo aver inserito la direttiva #include <ctime> si può utilizzare l'espressione time(0) nel proprio programma, ogni volta che viene valutata, tale espressione ritorna un numero intero uguale al numero di secondi trascorsi dal 1 gennaio 1970

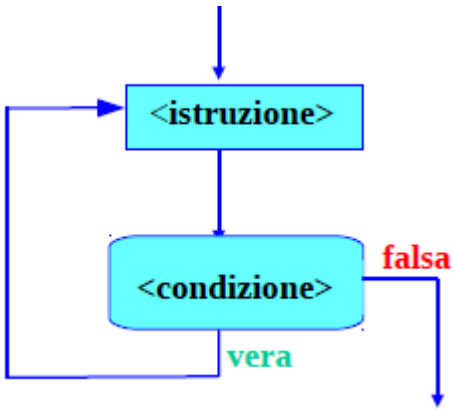
In caso di ciclo infinito, come si termina un programma? Ctrl + C manda il segnale DIGINTR (interruzione al processo di esecuzione, ma a volte può non bastare per interrompere il processo, chiudere il terminale potrebbe chiudere il processo, se così non fosse però: ad ogni processo è associato un identificatore numerico: pid, per elencare in propri processi dalla shell ps x, per elencare tutti i processi ps ax, cercare il processo con lo stesso nome dell'eseguibile.

Comando kill: spedisce segnali ai processi, per uccidere un processo : kill -9 <pid> (funziona sempre!!).

Spesso una struttura dati viene utilizzata per **rappresentare i dati del problema reale,** o almeno la parte necessaria per permettere all'algoritmo di risolverlo.

DO….WHILE

Do <istruzione> while (<condizione>)




È una variazione dell'istruzione While, con la differenza che la condizione viene controllata DOPO aver eseguito <istruzione>; quindi il ciclo viene **sempre eseguito almeno una volta.**

Come nel while modificare prima o poi la condizione per evitare il ciclo infinito

Si esce dal ciclo quando la condizione è falsa (come nel while)

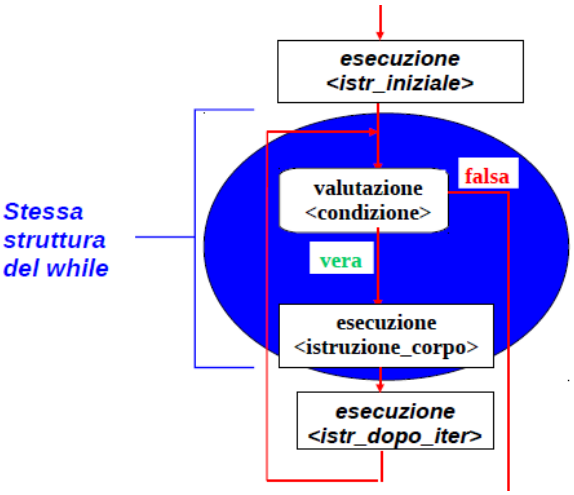
Adatto per controllare i valori in input (es: n deve essere positivo per andare avanti
do
cin>>n;
while (n<=0);)

Consiglio: scrivere la condizione al contrario può confondere, può aiutare scrivere la condizione che si vuole che sia vera per uscire dal ciclo e poi metterci un ! davanti

FOR è un'estensione dell'istruzione while in cui sono previsti, oltre al punto in cui inserire la condizione del ciclo, anche un punto in cui inserire l'istruzione da eseguire **subito prima** della prima iterazione ; e un punto in cui inserire l'istruzione da eseguire **subito dopo** ciascuna iterazione

For (<istr_iniziale>; <condizione>; <istr_dopo_iter>)

<istruzione_corpo>




<Istr_iniziale> tipicamente utilizzata per inizializzare i valori delle variabili, mentre <istr_dopo_iter> per modificare le variabili che determinano la condizione del ciclo.

Si può inserire una **definizione** (di variabile) **come istruzione iniziale nell'intestazio**ne del for, in questo modo la variabile si può utilizzare solo nell'intestazione e nel corpo del ciclo for.

Si possono inizializzare più variabili nell'istruzione iniziale del for e se ne possono fare di più eseguire dopo la fine di ogni interazione, basta separare tutto con una virgola

Le generiche espressioni <espr1>, <espr2>,…, <esprN> si possono concatenare mediante la virgola per ottenere l'espressione composta; le espr verranno valutate l'una dopo l'altra

Mentre per l'inizializzazione multipla nell'intestazione del for, tutte le variabili devono essere dello stesso tipo.

Sia le due istruzioni che la condizione, previste nell'intestazione del ciclo for possono essere omesse, il separatore (; ) deve rimanere, e se manca la condizione la si assume sempre vera.

For equivalente al while: for ( ; <condizione>; ) <istruzione>

For che genera un ciclo infinito: for ( ; ; ) <istruzione>

Il corpo di un ciclo può a sua volta contenere altri cicli, si chiama quindi ciclo annidato

Ci sono due istruzioni senza argomenti che permettono:

1) di uscire immediatamente da un ciclo :
   Break; (è la stessa istruzione già vista per lo switch)
2) di modificare la normale sequenza di esecuzione di una iterazione:
   continue; (utilizzabile solo in un ciclo)

BREAK provoca l'immediata uscita da un ciclo o dal corpo di uno switch; il break fornisce quindi un'uscita alternativa da un ciclo oltre la valutazione della condizione, più flessibile, ma aumenta la difficoltà di comprensione del programma.

CONTINUE si può usare solo nel corpo di un ciclo e fa saltare alla fine di questo come se fosse un salto alla parentesi } che chiude il blocco, quindi causa una nuova valutazione della condizione del ciclo e l'eventuale inizio della prossima iterazione

Istruzione vuota : è un semplice ";", non fa nulla ma può tornare utile nel ciclo for se l'intestazione fa già delle istruzioni.

(RI)Cominciamo bene (consigli pratici per la programmazione):

- Non  inseriamo numeri senza nome
- Inseriamo piuttosto sempre delle costanti con nome
- Usiamo nomi significativi per le variabili
- Non replichiamo codice, la replicazione rende il codice lungo e difficile da capire, aumentando inoltre la probabilità di commettere errori
- Analizziamo bene le consegne

Manipolatori= oggetti che se passati allo stream di uscita ne modificano il comportamento; possono essere persistenti: influenzano tutte le successive scritture.

Es: manipolatore hex, se passato alla cout da lì in poi tutti i numeri saranno stampati in base 16, per tornare alla base 10 : manipolatore dec.

Per usare la libreria matematica aggiungere -lm al g++ e includere la libreria matematica con #include <cmath> per poter usare operazioni tipo la radice quadrata di un numero: static_cast<int>(sqrt(N))

Prefissi binari dedicati per grandezze:

- 1024 Ki kibi
- 1024*Ki Mi mebi
- 1024*Mi Gi gibi
- 1024*Gi Ti tebi
- 1024*Ti Pi pebi
- 1024*Pi Ei exbi
- 1024*Ei Zi zebi
- 1024*Zi Yi yobi

## Lezione 06

Per conoscere come sono rappresentate le informazioni in un calcolatore bisogna conoscere la rappresentazione dei numeri in base 2, perché le informazioni sono rappresentate come sequenza di bit, ossia cifre con due possibili valori.

Rappresentazione di un numero in una data base: sequenza di cifre

Cifra= simbolo rappresentante un numero

Base= numero (naturale) di valori possibili per ciascuna cifra

Es in base 10 le cifre sono:

0 che rappresenta il valore 0

1 che rappresenta il valore 1

2 che rappresenta il valore 1+1

3 che rappresenta il valore 1+1+1

.. 9 che rappresenta il valore 1+1+1+1+1+1+1+1+1

Rappresentazione di un numero su n cifre in base b

ES: b=10, a(i) = {0,1,2,….,9}

345 => a2=3, a1=4, a0=5.

Base 2, 2 cifre: 0,1

La cifra nella posizione i-esima ha peso 2^i

Esempi (configurazioni di bit) :

base 10: 0= base 2:0

1=1

2=10=[1*2+0*1]

3=11=[1*2+1*1]

Una base molto spesso conveniente è la base 16, 16 cifre: 0,1,2,….,9,A,B,C,D,E,F, Esempi:

0=0, 10(base10)=A(in base 16), 18 (in base 10)= 12 (in base 16)= [1*16+2*1].

Per passare dalla base 2 a 16 si divide la sequenza base in sotto-sequenza consecutive da 4 cifre ciascuna partendo da destra, convertendo poi ciascuna sequenza di 4 cifre nella corrispondente cifra in base 16.

Esempio: dato il numero $[1000001111]_2$ , dividiamo le cifre in gruppi da quattro da destra 10 0000 1111, aggiungiamo due zeri all'inizio (senza modificare il valore del numero): 0010 0000 1111, in base 16 otteniamo 2 0 F; viceversa data la rappresentazione in base 16 di un numero naturale, il corrispondente numero in base 2 si ottiene convertendo semplicemente ciascuna cifra della rappresentazione in base 16 nella corrispondente sequenza di 4 cifre in base 2.

In una cella di memoria si può memorizzare con facilità un numero naturale memorizzando la sua rappresentazione in base 2, per rappresentare i numeri positivi e negativi (non vogliamo perdere un bit per il segno) quindi sommiamo una costante ai numeri negativi, per far si che diventino positivi, in modo che i VERI numeri positivi cadano in un intervallo di valori diverso da quello in cui cadono i FALSI numeri positivi, quindi si ottiene che si possono rappresentare generici numeri n (positivi o negativi) contenuti nell'intervallo $[-2^{N-1}, 2^{N-1}-1]$.

Una sequenza di N bit che rappresenta un numero intero negativo ha sempre il bit più a sinistra uguale a 1.

Gli oggetti di tipo **int** sono tipicamente rappresentati in complemento a 2.

LEZIONE 7

Abbiamo già visto un'istruzione composta, aggiungiamo che può anche venire chiamata **blocco.**

Cosa rimediamo grazie alle funzioni?

- Replichiamo codice
- Scarsa leggibilità

In presenza di un programma lungo e che si ripete in ciò che fa, per migliorarlo possiamo: aggiungere commenti per ottenere un po' di leggibilità, ma l'ideale sarebbe dare un nome significativo ad un pezzo di codice e chiamarlo per nome; in questo caso una funzione is_prime() la quale ci dice se un numero è primo o no (lo si capisce bene dal nome significativo) migliora la leggibilità del programma e avendo scritto il codice una volta sola nella funzione, viene eliminata la replicazione.

FUNZIONI:

l'astrazione di funzione è presente in tutti i linguaggi di programmazione di alto livello.

Come nel concetto matematico le funzioni prevedono **molti ingressi** (**detti parametri**), ed **una sola uscita**(risultato o valore di ritorno)

Inoltre il nome della funzione è tipicamente una parola (es: fattoriale(n))

Uso della funzione: **chiamata o invocazione:**

l'esecuzione di una funzione consiste nell'esecuzione di un frammento di codice; per avviare l'esecuzione di una funz bisogna eseguire un'invocazione o chiamata della funzione.

A seguito dell'invocazione si esegue il codice della funzione, dopodichè il controllo torna all'istruzione successiva all'invocazione stessa.

Definizione e dichiarazione

Una definizione di funzione è costituita da una **intestazione** e da un **corpo**, definito mediante un blocco (istruzione composta)

Esempio di intestazione: int fattoriale (int n) → int = tipo del risultato, fattoriale= nome della funzione (int n) = lista dei parametri formali.

Intestazione di una funzione di nome **fattoriale** che prende in ingresso un valore di tipo **int** e ritorna **un valore** di tipo **int**.

Parametro formale (nel nostro caso è tipo int e chiamato n)= significa che quando la funzione verrà invocata, le dovrà obbligatoriamente essere passato un valore, tale dovrà essere memorizzato nel parametro formale.

Quando una funzione "Ritorna un valore" non significa su cout, ma vuol dire che, se il nome della funzione appare in una espressione, allora quando l'espressione viene valutata, il nome della funzione sarà sostituito dal valore di ritorno della funzione stessa.

Es: int somma (int x, int y) → la funzione prende in ingresso **due** valori di tipo int e ritorna **un** valore di tipo int

Void stampa_n_volte (int n) → non ritorna nulla (void) tipo di ritorno vuoto, e non si potrà usare in un'espressione.

Void stampa_2_volte (void) → non prende in ingresso nulla e non ritorna nulla (sarebbe come scrivere void stampa_2_volte())

L'intestazione specifica nell'ordine:

1) **Tipo del valore di ritorno** => (void se non c'è risultato: corrisponde alla procedura di altri linguaggi )
2) **Nome della funzione**
3) **Lista dei parametri formali** (in ingresso)

I parametri formali sono visibili, possiamo cioè essere utilizzati all'interno del corpo della funzione come variabili normali

Una funzione **non può essere definita all'interno di un'altra funzione!!**

Uso di una funzione: **chiamata o invocazione** (seconda parte)

Una chiamata è costituita dal **nome della funzione** e dalla **lista dei parametri attuali tra parentesi tonde:**

<nomeFunzione> (<lista-parametri-attuali>)

Un parametro attuale può essere una qualsiasi **espressione**.

L'istruzione più semplice che contenga una chiamata è la seguente : <nomeFunzione> (<lista-parametri-attuali>) ;

l'effetto di tale istruzione è quello di far partire l'esecuzione della funzione, una volta terminata si riprende dall'espressione successiva a quella dell'invocazione.

L'invocazione di una funzione comporta i seguenti passi:

1) **Calcola** il valore di ciascuno dei parametri attuali
2) **Definisce** ciascun parametro formale della funzione e lo **inizializza** con il valore del parametro attuale che si trova nella stessa posizione
3) Si esegue la funzione

In una chiamata di funzione si dovranno inserire tanti parametri formali della funzione, ogni parametro formale della funzione sarà **inizializzato** con il parametro attuale nella stessa posizione nella chiamata, prima di iniziare ad eseguire la funzione stessa

La corrispondenza tra parametri formali e attuali è **posizionale**, con in più il controllo di tipo.

> Si presume che la lista dei parametri attuali abbia lo stesso numero di elementi, e che il tipo di ogni parametro attuale sia compatibile con il tipo del corrispondente parametro formale.

La corrispondenza tra i nomi dei parametri attuali e formali **non ha nessuna importanza**

> Conta solo la posizione all'interno della chiamata

Una chiamata di una funzione è una **espressione** e si può inserire a sua volta in una espressione, a patto che il suo tipo di ritorno non sia void

Nel caso non sia void, il valore di ritorno della funzione costituisce un fattore dell'espressione, la funzione è invocata quando bisogna calcolare il valore di tale fattore, il valore del fattore sarà uguale al valore di ritorno della funzione; al termine della chiamata, riprende il calcolo del valore dell'espressione in cui la chiamata di funzione è inserita.

Come si stabilisce il valore di ritorno di una funzione? → istruzione **return**

Viene usata per far terminare l'esecuzione della funzione e far proseguire il programma dall'istruzione successiva a quella con cui la funzione è stata invocata

Ossia per restituire il controllo alla funzione chiamante.

Se la funzione ha tipo di ritorno diverso da void, è mediante l'istruzione **return** che si determina il valore di ritorno della funzione

Sintassi nel caso di funzione void: return;

sintassi nel caso di funzioni con tipo di ritorno diverso da void: return <espressione>;

il tipo del valore dell'espressione deve coincidere col tipo del valore di ritorno specificato nell'intestazione della funzione (o essere perlomeno compatibile)

Eventuali istruzioni dopo il **return** non saranno eseguite!

Nel caso della funzione main, l'esecuzione dell'istruzione return fa uscire dall'intero programma

Una funzione di tipo void può terminare o con il return o quando l'esecuzione giunge in fondo alla funzione.

Al contrario funzioni NON void devono avere sempre un return.

Una funzione può essere invocata solo da un punto del programma successivo , nel testo del programma stesso, alla definizione della funzione.

Definizione e dichiarazione della funzione (seconda parte)

Definiamo come **locale ad una funzione** un oggetto che si può utilizzare solo all'interno della funzione

Un parametro formale è un oggetto locale di una funzione.

Nel caso sia variabile, il suo valore può essere modificato all'interno della funzione, nel caso sia costante, il suo valore, utilizzato all'atto della chiamata della funzione, non può essere cambiato.

Variabili e costanti definite all'interno del corpo sono locali alla funzione e se non inizializzate (le variabili locali) **hanno valori casuali.**

[dalla sesta esercitazione]

Per creare un programma efficiente bisogna cercare di provare il programma per tutti gli ingressi possibili o almeno per un'alta percentuale.

Testing a scatola aperta (white box):

> mi metto nei panni del compilatore prima e soprattutto dell'esecutore dopo, cerco di capire come vanno le cose al variare dei rami di codice eseguiti.

Testing a scatola chiusa (black box)

> Si opera sui valori di ingresso supponendo di non sapere nulla di come funziona il programma, si provano i valori agli estremi, nel mezzo, fuori dagli estremi degli intervalli consentiti

Esistono due tipi di fallimenti del programma:

1) Il programma viene terminato forzatamente dal sistema operativo (es: divisione per 0)

2) Il programma non viene terminato, ma fornisce risultati scorretti.

Una volta scoperta la presenza di errore(chiamato bug o baco) il passo successivo è trovare l'errore = debugging.

Prima cosa: cercare di capire dove sta l'errore; spesso si trovano in prossimità delle variabili (in lettura o scrittura) e per trovarlo si effettua del *tracing* = per indicare la stampa di valori o in generale messaggi per capire cosa sta facendo esattamente un programma.

In un ciclo conviene inserire uno stdin per controllare il ritmo delle iterazioni durante l'esecuzione.

Torniamo alle lezioni.

In una funzione conviene sempre far entrare variabile come parametro formale e non chiederlo in ingresso nella funzione stessa, perché la funz può essere usata ovunque all'interno del programma passandole il valore che si preferisce, non è necessario dover leggere obbligatoriamente qualcosa da stdin (eventuali letture da stdin si possono effettuare semplicemente dal main ).

Inoltre in una funzione conviene far restituire un valore alla funzione attraverso l'istruzione return, perché la funz può essere utilizzata ovunque all'interno del programma, non è necessario stampare qualcosa su stdout ed eventualmente si possono fare semplicemente nel main

Il main è una funzione:

- La prima istruzione nel main è la prima dell'intero programma
- Le variabili definite al suo interno hanno valore casuale
- Terminato il main, termina tutto il programma
- In un programma corretto il main ritorna un tipo int
- Il valore ritornato dal main coincide col valore restituito dal processo quando termina

Una definizione è un caso particolare di dichiarazione, una def di variabile o costante è una dichiarazione che causa l'allocazione di spazio in memoria quando viene incontrata, una def di funz è un caso particolare di dichiarazione in cui si definisce il corpo della funzione.

Una dichiarazione è un'istruzione in cui si introduce un nuovo identificatore e se ne dichiara il tipo.

Dichiarazione (senza definizione) o **prototipo** di una funz è costituita dalla sola intestazione di una funzione seguita da un punto e virgola : <nomeTipo> <nomeFunzione> (<lista-parametri>);

il prototipo:

- È un puro "avviso ai naviganti"
- **Non causa la produzione di alcun byte di codice eseguibile**
- Può essere ripetuto più volte nel programma (basta che non ce ne siano due in contraddizione)
- Può comparire anche dentro un'altra funzione (non lo useremo)

La definizione:

- Contiene il codice della funzione
- Non può essere duplicata!
- Non può essere inserita in un'altra funzione
- Il nome dei parametri formali, non necessario in un prototipo, è importante in una definizione

Quindi il prototipo di funzione può comparire più volte, ma la funz deve essere definita una sola volta.

Passaggio dei parametri= è l'inizializzazione dei parametri formali di una funzione mediante i parametri attuali, che avviene al momento della chiamata della funzione

Passaggio per valore: le locazioni di memoria corrispondenti ai parametri formali:

- Sono allocate al momento della chiamata della funzione
- Sono inizializzate con i valori dei corrispondenti parametri attuali trasmessi dalla funzione chiamante
- Vivono per tutto il tempo in cui la funzione è in esecuzione
- Sono deallocate quando la funzione termina

QUINDI la funz chiamata effettua una copia dei valori dei parametri attuali passati dalla funzione chiamante, le copie sono private, e ogni modifica è strettamente locale alla funzione, i parametri attuali della funzione chiaramente non saranno mai modificati!

In generale è cattiva abitudine modificare i parametri formali per utilizzarli come variabili ausiliarie, poca leggibilità e crea effetti collaterali nel caso di passaggi per riferimento

L'unico caso in cui è necessario ed appropriato modificare i parametri formali è quando tali parametri sono intesi come **parametri di uscita**= parametri in cui devono essere memorizzati valori che saranno poi utilizzati da chi ha invocato la funz.

Questo non può però accadere nel caso di passaggio di valore, perché i parametri formali sono oggetti locali alla funzione e quindi verranno eliminati alla terminazione della funzione stessa

Nei passaggi per valore, è sicuro che le variabili del chiamante e del chiamato **sono completamente disaccoppiate**, consentendo di ragionare per componenti isolati

LIMITI:

impedisce a priori di scrivere funzioni che abbiano lo scopo di modificare variabili utilizzate poi nella funzione da cui sono invocate

come vedremo il passaggio per valore può essere costoso per dati di grosse dimensioni

LEZIONE 8

Dichiarazione globale= cioè all'esterno del corpo delle funzioni, in questo caso, se il programma sta su un solo file, si parla di dichiarazioni e di identificatori globali

Mentre identificatori dichiarati nei blocchi sono invece denominati locali (al blocco).

Una caratteristica di un oggetto può essere: **statica** se non cambia durante l'esecuzione del programma e **dinamica** se invece cambia, nel caso di variabili:

- Nome: definito con identificatore (statico)
- Indirizzo (lvalue): locazione di memoria a partire dalla quale è memorizzata la var (statico)
- Valore (rvalue) contenuto dell'area di memoria associata alla var (dinamico)
- Tipo: insieme dei valori che la var può assumere e di operazioni ad essi applicati (statico)
  Il C/C++ è un linguaggio con tipizzazione statica e forte (il compilatore controlla il rispetto del tipo)
  Il tipo di una variabile **non cambia** durante l'esecuzione del programma.

**Campo di visibilità (scope**) di un identificatore= parte (del testo ) in cui l'identificatore può essere usato

**Tempo di vita di un oggetto:** intervallo di tempo in cui l'oggetto è mantenuto in memoria (in C/C++ tempo di vita è **dinamico,** ovvero gli oggetti possono nascere e morire durante il programma).

Regole di visibilità:

**regole di visibilità statiche**: il campo di vis è stabilito in base al testo del programma, cioè dove sono dichiarati gli identificatori (come in C/C++, java, pascal)

**regole di visibilità dinamiche:** il campo di visibilità è stabilito dinamicamente in base alla sequenza di chiamata delle funzioni.

Riassumendo in C/C++ il tempo di vita degli ide dipende da dove sono collate le corrispondenti dichiarazioni nel programma, consideriamo 3 casi:

1) Ide definiti in un blocco che definisce il corpo di una funzione diversa dal main
2) Ide dichiarati nel blocco del main
3) Ide globali

I primi due casi sono **identificatori locali ad un blocco**

Caso 1 (blocco che non coincide con il corpo del main): Il **campo di visibilità** degli ide dichiarati in un blocco è il blocco stesso, Dal punto in cui sono dichiarati fino alla fine del blocco stesso.

Eccezion fatta che se nel blocco compare un blocco innestato in cui è dichiarato un identificatore con lo stesso nome, l'oggetto del blocco esterno rimane in vita, ma l'identificatore non è visibile nel blocco innestato.

Il **tempo di vita** va dal momento della definizione fino alla fine dell'esecuzione delle istruzioni del blocco (senza eccezioni)

Caso 2 (identificatori dichiarati nel corpo del main): hanno **tempo di vita pari alla durata del programma**, ma non sono visibili in qualunque parte del programma= sono visibili solo dal punto in cui sono dichiarati fino alla fine del corpo del main, con l'esclusione di blocchi annidati in cui siano dichiarati identificatori con lo stesso nome.

Caso 3 (identificatori globali) Sono visibili dal punto in cui sono stati dichiarati fino alla fine del file, con l'eccezione di un ide dichiarato con lo stesso nome in un blocco, l'oggetto rimane in vita, ma non è visibile nel blocco innestato.

Hanno però **tempo di vita pari alla durata dell'intero programma.**

Valori iniziali variabili: dipende da dove tale variabile è definita → all'interno di un blocco (main o di una funzione) **valore iniziale casuale,** al di fuori dei blocchi **valore iniziale 0**.

Dipendenza= tra due parti c'è dipendenza se il funzionamento di una delle due parti dipende dalla presenza e dal corretto funzionamento dell'altra parte; quindi nostro obbiettivo è ridurre al minimo le dipendenze.

Gli oggetti globali aumentano la complessità dei programmi, creano molto effetto di dipendenza; al contrario gli oggetti locali riducono la complessità dei programmi ( bisogna quindi preferirli a quelli globali).

LEZIONE 9

Si dice che un operatore logico binario è **valutato in corto circuito** se il suo secondo operando non è valutato se il suo valore del primo operando è sufficiente a stabilire il risultato (in C/C++ sono valutati in corto-circuito gli operatori && e ||).

ES: false && x => il primo è sufficiente a dire che è falso e il secondo non è valutato

True || f(x) => il primo è già vero, quindi il secondo non viene valutato e f(x) non viene invocata

22 || x => il primo è già vero

&& e || sono associativi a sinistra, quindi a && b && c == ( a && b ) && c (se a&&b è falso, c non viene valutato)

**Espressione condizionale=** <condizione>?<espressione1>:<espressione2>

Se condizione è vera, si usa <espressione1>, altrimenti 2.

**Tipi di dato primitivi=> Caratteri (char)**

Rappresenta l'insieme dei caratteri utilizzabili in accordo allo standard del linguaggio C/C++

Costanti letterali carattere= dato un carattere, la corrispondente costante letterale si ottiene racchiudendo il carattere tra singoli apici 'a' 'd' '@' ; da qualche parte nel programma è stato memorizzato tale carattere, che è stato poi passato in qualche modo dal programma al terminale, ma sappiamo che la memoria può contenere (solo) numeri, possibile soluzione **associare un numero intero, ossia un codice, diverso a ciascun carattere**; generalmente si usa il codice ASCII codifica che usa un byte, quindi 256 codici.

Il tipo char non denota un nuovo tipo in senso stretto, ma è di fatto l'insieme dei valori interi rappresentabili su di un byte char = 1 byte = -128 a 128 ( se consideriamo il segno) oppure da 0 a 255; unsigned char 1 byte da 0 a 255.

Le costanti carattere **denotano dei numeri interi:** scrivere una costante caratteri equivale a scrivere il numero corrispondente al codice ASCII del carattere (es: 'a' è equivalente a scrivere 97) però **una costante carattere ha anche associato il tipo char** ovvero, se scrivessimo cout<<'a'<<endl; abbiamo passato 97 al cout, ma non stampa 97, bensì il carattere "a" perché l'operatore << ha dedotto dal tipo (char) cosa fare.

I caratteri sono ordinati: 1<2<…<9 A<B<C<……<Z a<b<c<…..<z

Ma qual è l'ordinamento tra le classi? (es '1'<'a'?) non è definito, tra le classi lo standard del linguaggio non prevede nessuna garanzia di quale dei possibili ordinamenti viene adottato l'ASCII ha il suo ordinamento, ma quale codifica deve essere utilizzata non è definita dallo standard, essa è libera, basta che sia rispettata all'interno delle classi

Conversione di tipo

Dato il valore di una costante, var, funz, o espressione, tale valore ha associato anche un tipo, esiste un modo per convertire un valore di un tipo in quello di un altro, tramite la **conversione esplicita** (es da 97 di tipo in a 97 di tipo char, ovvero 'a').

Tre forme:

1) Cast (<tipo_di_destinazione>) <espressione> es: d=(int)a;
2) Notazione funzionale <tipo_di_destinazione> (<espressione>) es: d=int(a);

3) Operatore static_cast static_cast<<tipo_di_destinazione>>(<espressione>)
es: d=static_cast<int>(a)

In tutti e tre i casi, il valore dell'espressione è convertito nel corrispondente valore di tipo <tipo_di_destinazione>, quale che sia il tipo del valore dell'espressione.

L'uso dell'operatore static_cast comporta una notazione più pesante rispetto agli altri due, ma ciò è voluto perché le conversioni sono spesso pericolose, ma con static_cast il compilatore usa regole più rigide (programma più sicuro).

Manipolatore noskipws= serve a configurare cin per non saltare gli spazi bianchi, viceversa, per tornare a saltarli si usa skipws

Dopo aver premuto un carattere stampabile diverso dall'invio, il programma non riesce a leggerlo perché è configurato in maniera cosiddetta **canonica**, ovvero il terminale manda i caratteri allo stdin del programma solo dopo che è stato premuto invio, si può configurare il terminale in modo **non canonico** e farsi inviare ogni carattere non appena viene immesso dall'utente (non ha rilevanza con l'esame).

Operazioni : sono applicabili tutti gli operatori visti per il tipo int (/,<,-,+,ecc).

**Portabilità:** un codice è portabile se ciò che scriviamo non fa nessuna assunzione su come sono rappresentati i numeri

Al contrario, un codice non è portabile se il codice fa affidamento sul fatto che i numeri sono rappresentati in un certo modo, allora se eseguito su una macchina con numeri rappresentati in modo diverso, quel programma non funziona più.

LEZIONE 10

Tipi di dato Enumerati (enum)

Possono essere utili quando i valori delle variabili sono **solo** valori accettabili dalla circostanza (es: giorni della settimana).

**Insieme di costanti** definito dal programmatore, ciascuna individuata da un identificatore (nome) e detta **enumeratore**, es di dichiarazione: enum colori_t {rosso, verde, giallo} ; dichiara un tipo enumerato di nome "colori_t" e tre cost. intere (enumeratori) di nome "rosso, verde e giallo"; gli oggetti di tipo colori_t possono assumere solo quei 3 enumeratori, agli enumeratori sono assegnati numeri interi consecutivi a partire da zero, a meno di inizializzazioni esplicite.

Mediante il tipo colori_t sarà possibile definire nuovi oggetti mediante delle definizioni, con la stessa sintassi usata per i tipi predefiniti (così come si può scrivere "int a;", si potrà scrivere "colori_t a;" stiamo definendo quindi un oggetto di nome a e di tipo colori_t, i valori possibili di oggetti saranno le costanti rosso, verde e giallo.

Agli enum son associati per default valori interi consecutivi a partire da 0.

ATTENZIONE!! Se si dichiara una variabile o un nuovo enumeratore con lo stesso nome di uno già dichiarato, da quel punto in poi si perde la visibilità del precedente enum.

Si possono inoltre inizializzare le costanti, e si può definire una variabile di tipo enum senza dichiarare il tipo a parte (es: esum {<lista_dich_enumeratori>} <identificatore>; )

Lo spazio occupato in memoria da un oggetto enum dipende dal compilatore (tipicamente stessa di int).

Non si effettuano mai operazioni tra enum e oggetti di altro tipo, inoltre il compilatore aiuta il programmatore a non assegnare valori impropri ad un oggetto di tipo enum (ad es: colore_t c=100 causa un errore a tempo di compilazione)

Sono però lecite ma pericolose operazioni di static_cast.

Enum class (dichiarazione uguale all'enum normale), la differenza con enum normale sta nell'utilizzo dell'enumeratore, bisogna infatti aggiungere come prefisso il nome del tipo seguito da :: (es: cout<<colore2_t::blu; ) questo permette a due o più enumerati di avere gli enumeratori con lo stesso nome senza che sorgano problemi.

Altro vantaggio è che non è possibile alcuna delle operazioni pericolose del tipo enum; si può inoltre decidere il tipo di dato utilizzato per memorizzare i valori degli enumeratori e quindi decidere quanta memoria viene occupata (non la vedremo in questo corso).

Benefici enum: migliore leggibilità, indipendenza del codice dai valori esatti e dal numero di costanti, maggiore robustezza agli errori.

NUMERI REALI

In C++ si possono usare notazione normale (14.3) e scientifica (10^3)

L'elaboratore potrebbe stampare un numero che non coincide con quello in memoria dato che stampa solo un numero di cifre dopo la virgola ragionevole.

Anche i numeri reali sono memorizzati sotto forma di sequenza di bit, ma rappresentati in due modi:

1) **Virgola fissa**: numero massimo di cifre intere e decimali deciso a priori (es 3 cifre per la parte intera e 2 per quella decimale)
2) **Virgola mobile:** numero massimo totale di cifre, intere e decimali, deciso a priori, ma posizione della virgola libera (es: 5 cifre totali)

La scelta della virgola mobile conviene perché permette una rappresentazione abbastanza semplice dei numeri in memoria e operazioni più veloci; un numero reale è memorizzato mediante 3 componenti:

1) **Segno**
2) **Mantissa (**le cifre del numero)
3) **Esponente** in base 10

La mantissa è immaginata come un numero virgola fissa, con la virgola posizionata sempre subito prima della prima cifra diversa da 0

La mantissa di un numero reale si ottiene semplicemente spostando la posizione della virgola del numero di partenza.

Se la mantissa è ottenuta spostando la virgola di n posizioni verso sinistra, allora l'esponente è uguale ad n se viceversa è ottenuta spostando la virgola di n posizioni verso destra, l'esponente sarà -n.

In C++ i num reali sono rappresentati mediante i tipi float e double:

sono numeri in virgola mobile che mirano a rappresentare (con diversa precisione) un sottoinsieme di numeri reali; entrambi sono un'approssimazione sia per **precisione** (numero di cifre della mantissa) sia come **intervallo** di valori rappresentabili.

Nel caso di un numero con tanti decimali o periodico non vengono stampate tutte le cifre ma si può utilizzare un manipolatore: setprecision(<numero_cifre>) setta in numero massimo di cifre per un numero in virgola mobile; bisogna includere <iomanip>, l'effetto è persistente ovvero dura fino alla prossima eventuale chiamata di setprecision.

Come rappresentare esponenti di valore negativo? Così come per la memorizzazione di numeri negativi nei bit, si somma l'esponente ad un numero intero predefinito (offset) detto ciò, l'esponente effettivo è il risultato di num-offset.

Definiamo **precisione P** di un dato numerico in una base b come: il numero massimo di cifre in base b tali che qualsiasi numero rappresentato da P cifre è rappresentabile in modo esatto con tale tipo di dato .

Valori tipici per float e double: float precisione 6 cifre decimali
intervallo di valori assoluti = $3,4*10^{-38}$…$3,4*10^{38}$

Double 15 cifre decimali, int di valori ass = $1,7*10^{-308}$…..$1,7*10^{308}$

Occupazione di memoria float 4 byte, double 8 byte, long double 10 byte.

Non tutti i numeri reali sono rappresentabili siccome gli operatori hanno precisione limitata e per i numeri non rappresentabili si **utilizza l'esponente**

Se un numero reale ha più cifre di quelle che si possono rappresentare, avviene un **troncamento**.

Ogni numero frazionario (<1) che è rappresentato da una qualsiasi sequenza di cifre dopo la virgola in base 2, è uguale alla somma di numeri razionali con una potenza di 2 al denominatore (uno per ogni cifra); quindi solo i numeri razionali frazionari che hanno potenza di 2 al denominatore si possono esprimere con una sequenza finita di cifre binarie. ( parte da rivedere)

Conversione da reale ad intero: tipicamente fatta per troncamento, il valore convertito dovrà appartenere a qualcuno dei tipi numerabili.

Operazioni tra reali ed interi: se si esegue una operazione tra un oggetto di tipo int, enum o char ed un oggetto reale, si effettua di fatto la variante reale dell'operazione.

Lo 0 in float e double è rappresentato in modo approssimato (la mantissa ha per definizione la prima cifra diversa da 0).

Possono verificarsi errori dovuto al troncamento, quindi meglio evitare l'uso dell'operatore ==, usare piuttosto <= e >=.

TIPI E CONVERSIONI DI TIPO

Tipi interi: int 32 bit, short int 16 bit, long int 64 bit

Tipi naturali : unsigned int 32 bit, unsigned short int 16 bit, unsigned long int 64 bit.

Un oggetto unsigned ha solo valori maggiori o uguali a 0.

Da C++ 11 esiste anche il tipo long long int.

Tipo carattere : char 8 bit, signed char 8 bit, unsigned char 8 bit.

Tipo reale : float 32 bit, double 64 bit, long double 80 bit.

Tipo booleano: bool.

Tipo enumerato: enum e enum class.

Mediante suffissi si possono scrivere espressioni letterali dei seguenti tipo: U→unsign. Int; UL→unsign. Long; ULL→ unsign long long.

Espressioni eterogenee

Fin quando gli operandi di un operazione sono dello stesso tipo (**omogenei)** non ci sono dubbi sul comportamento, mentre se gli operandi sono di tipi diversi siamo in presenza di un tipo **eterogeneo**, in generale definiamo eterogenea una espressione che contenga fattori o termini di tipo eterogeneo.

In presenza di operandi eterogenei ci sono due possibilità:

1) Il programmatore inserisce **conversioni esplicite,** per omogeneizzare.
2) Il programm. Non inserisce conv. Esplicite, in questo caso se possibile, il compilatore effettua **conversioni implicite** (**coercion)** oppure fallisce la compilazione.

Il C/C++ è un linguaggio a *tipizzazione forte*: il compilatore controlla il tipo degli operandi di ogni operazione per evitare operazioni illegali, per tali tipi di dato o perdite di informazione.

Le conversioni implicite di tipo che non provocano perdita sono fatte dal compil. Senza dare alcuna segnalazione; tuttavia le conversioni implicite che possono provocare perdita di info **non sono illegali**, vengono tipicamente segnalate mediante **warning.**

Regole utilizzate in presenza di **operandi eterogenei per un operatore binario diverso dall'assegnamento.**

Ogni operando di tipo char o short viene convertito in INT. Se dopo questa operazione gli operandi sono ancora eterogenei si continua a convertire seguendo questa gerarchia: CHAR < INT < UNSIGNED INT < LONG INT < UNSIGNED LONG INT < FLOAT < DOUBLE < LONG DOUBLE.

A questo punto i due operandi sono omogenei e viene invocata l'operazione relativa all'operando di tipo più alto.

L'espressione a destra dell'assegnamento viene valutata come descritto dalle regole per la valutazione di un'espressione omogena viste finora.

Se il **tipo del risultato** è diverso da quello della variabile a sinistra dell'assegnamento, allora viene **convertito al tipo di tale variabile**.

- Se il tipo della variabile è gerarchicamente uguale o superiore al tipo del risultato dell'espressione, tale risultato viene convertito al tipo della variabile probabilmente senza perdita di informazioni
- Se il tipo della variabile è gerarchicamente inferiore al tipo del risultato dell'espressione, tale risultato viene convertito al tipo della variabile con alto rischio di perdita di informazioni.

Se memorizziamo un numero senza cifre dopo la virgola all'interno di un oggetto di tipo double, e assegniamo il valore di tale oggetto in un altro di tipo int, si potrebbe avere perdita di informazioni perché l'oggetto di tipo int potrebbe non essere in grado di rappresentare tutte le cifre e quindi il rischio di overflow.

Se invece nel caso contrario passassimo da int a float, si avrebbe anche qui perdita di info, ci sarebbe il rischio che il float non sarebbe in grado di rappresentare tutte le cifre, si avrebbe un troncamento delle cifre del numero

Concludendo, le conversioni sono praticamente sempre pericolose.

Per terminare forzatamente l'esecuzione di un programma si usa la funzione "void exit (int n)" il valore passato come parametro attuale corrisponde al valore che sarà ritornato a chi ha fatto partire il programma.

Lezione 11

Per superare i limiti della semantica per copia, occorre consentire alla funzione di far riferimento alle variabili di chi le invoca, serve il concetto di passaggio per **riferimento** = se una funzione dichiara nella sua intestazione, che un parametro costituisce un riferimento , allora è un riferimento alla variabile passata come parametro attuale nella funzione chiamante; quindi → ogni modifica fatta al parametro formale in realtà viene effettuata sulla variabile della funzione chiamante passata come parametro attuale.

Un riferimento è = un identificatore associato all'indirizzo di un oggetto (quando si dichiara un oggetto, gli si assegna già un riferimento di default, tale riferimento è quello che finora abbiamo chiamato "nome dell'oggetto"

Sintassi riferimento: <tipo_oggetto> & <identificatore> = <nome_oggetto>;

l'inizializzazione di un riferimento all'atto della sua definizione è obbligatoria, una volta definito ed inizializzato, un riferimento si riferisce **per sempre allo stesso oggetto**.

Per passare un parametro attuale per riferimento basta definire il corrispondente parametro formale come un oggetto di tipo riferimento: <tipo_oggetto> & <identificatore> .

La sintassi del **passaggio** di un parametro attuale **per riferimento** all'atto di una chiamata di funzione è **identica a** quella del **passaggio per valore.**

Quindi: un parametro formale di tipo riferimento ha esattamente le caratteristiche che ci servono per realizzare il passaggio per riferimento precedentemente introdotto.

Tale parametro formale **non è più una variabile/costante locale**, ma è un riferimento all'oggetto passato come parametro attuale nella funzione chiamante, quindi ogni modifica fatta al parametro formale in realtà viene effettuata sull'oggetto della funzione chiamante passata come parametro attuale.

Sorge il problema di distinguere quando dalla chiamata di una funzione il passaggio è per valore o per riferimento (è uno dei pericoli di questo strumento).

Direzione dei parametri delle funzioni: schema puramente concettuale che poi si può realizzare, a seconda dei casi, con uno dei due tipi di passaggio; tipicamente si considerano tre possibilità:

1) Parametri di ingresso
2) Parametri di uscita
3) Parametri di ingresso/uscita.

Parametri di ingresso: oggetti utilizzati solo per fornire i valori di ingresso su cui deve lavorare la funzione

Parametri di uscita: oggetti che non saranno mai usati in lettura, ma solo per memorizzare dei valori di ritorno, permettono di fatto di definire funzioni con un vettore di valori di ritorno (uno viene ritornato mediante il **return**, gli altri con parametri di uscita)

Parametri di ingresso/uscita: usati come parametri di ingresso che come di uscita, realizzabili mediante passaggio per riferimento, tra i tre sono probabilmente il tipo di parametro che porta alla peggiore leggibilità.

A basso livello il compilatore realizza un riferimento mediante una variabile nascosta in cui memorizza l'indirizzo dell'oggetto di cui il riferimento è un sinonimo, es: int &a = b il compilatore memorizza in una variabile nascosta l'indirizzo di b, ed usa tale indirizzo tutte le volte che si deve accedere a b attraverso il riferimento a; pertanto, di fatto si passa solo l'indirizzo dell'oggetto.

Il **passaggio per valore** di oggetti molto grandi diviene tanto più costoso, sia in termini di tempo, sia di occupazione di memoria, conviene però passare un oggetto molto grande per riferimento rispetto a passarlo per valore (perché ha sempre lo stesso costo, indipendentemente dalla dimensione)

E per risolvere eventuali effetti collaterali, si aggiunge il qualificatore const.

Lezione 12 Ingegneria del codice

Il primo semplice principio è che il codice deve essere **formattato in modo opportuno**, abbiamo già visto l'indentazione, l'importante è sceglierne uno ed applicare sempre quello.

Usare **nomi significativi** per le variabili e le costanti, e per le **funzioni**.

Cercare di raggruppare concetti correlati e separare i gruppi tra di loro con opportuni spazi (righe bianche)

**Non replicare mai** uno stesso frammento di codice in più punti di un programma meglio utilizzare funzioni .

Man mano che il programma diventa più complicato, aumenta l'impegno in collaudo e correzione errori.

che criterio usare per scegliere i nomi? Devono essere **appartenenti al domino del problema.**

Usare il tipo di dato più adatto. **Progettare il cambiamento.**

**Complessità computazionale**: misura il costo di un algoritmo in termini di numeri di passi che deve compiere per ottenere l'obiettivo (definito anche costo computazionale).

Complessità intesa come quanto è difficile comprendere tale frammento di codice, ovvero il numero di oggetti che si devono tenere in mente per comprendere il frammento di codice (ad esempio abbiamo usato la funzione sqrt o l'operatore << senza ricordarci i dettagli interni).

Per evitare effetti collaterali minimizziamo 1) i passaggi per riferimento (senza il const) e 2) l'uso delle variabili globali.

Un frammento di codice è proporzionale al numero di punti di scelta presenti, ogni istruzione condizionale o iterativa comporta un punto di scelta, bisogna evitare di nidificare scelte e se proprio necessario non superare le 3 istruzioni nidificate, per evitare ciò basta prendere una o alcune delle istruzioni più interne e spostarle in una funzione.

I commenti sono **fondamentali,** migliorano leggibilità e manutenibilità; ma devono essere sintetici ma completi, un commento deve chiarire il frammento di un codice o fornire un riepilogo. È molto utile commentare le funzioni specialmente descrivere : lo scopo della funzione, dei parametri di ingresso, dei parametri di uscita, del valore di uscita e di eventuali effetti collaterali.

Lezione 13

Un Array è una ennupla di N oggetti dello stesso tipo allocati in posizioni in posizioni contigue in memoria.

Gli elementi vengono selezionati mediante **indice**, ciascun elemento dell'array è denotato mediante: nome dell'array seguito da un indice intero compreso fra 0 e N-1 scritto tra **parentesi quadre**.

Sintassi : <tipo_elementi_array> <identificatore> [espr_costante] ;

semantica: alloca una sequenza contigua di elementi in memoria, tutti di tipo <tipo_elementi_array> ed in numero pari ad <espr_costante> , esempio int vett[4].

All'atto della definizione di un array statico le dimensioni devono essere stabilite mediante una espressione costante=> il valore deve essere noto a tempo di scrittura del programma.

L'indice parte sempre da 0, pertanto un array di N elementi ha sempre indici da 0 a N-1 (inclusi).

All'atto della definizione di un array viene allocato spazio nella memoria del programma per una sequenza di celle, nel caso più semplice sono di un tipo di dato che sta in una sola cella di memoria, come accade per il tipo **char**, se così non fosse, l'array è una sequenza di sottosequenze di celle e ognuna di esso è utilizzata per memorizzare uno degli elementi dell'array.

Esistono array statici a lunghezza variabile, ma solo pochi compilatori e standard sono consentiti.

La sintassi di C/C++ non permette di utilizzare l'operatore di assegnamento per copiare il contenuto di un intero array all'interno di un altro array, l'unico modo è copiare gli elementi uno alla volta.

Di fatto l'array è l'oggetto che permette di implementare il vettore matematico.

La dimensione dell'array è implicitamente nascosta in una zona nascosa della memoria non controllabile dal programmatore, si può risalire alle dimensioni dell'array mediante l'operatore sizeof.

In c/c++ non è previsto nessun controllo della correttezza degli indici (es se un vett ha 100 elementi, nulla mi vieta di salvare un elemento in posizione 105, causerà fallimento a tempo di esecuzione).

Un array può essere inizializzato solo all'atto della sua definizione: <tipo_elementi_array> <identificatore> [<espr-costante] = {<espr1>,<espr2>,….,<esprN>}

Se un array è inizializzato, l'indicazione della dimensione si può omettere.

Per passare ad una funzione <tipo_elementi> <identificatore> [ ]

Es: void fun(char c, int v[], …) { … }

Gli array sono **automaticamente passati per riferimento** (per evitare ciò basta aggiungere il const)

La dimensione di un array non può essere ottenuta mediante sizeof , per comunicarlo ad una funzione deve essere il programmatore a passarglielo con due possibili soluzioni: un parametro addizionale da passare alla funzione o una var/const globale.

Vettori dinamici : la dimensione N di un array statico non può essere determinata o modificata a tempo di esecuzione, però varie applicazioni possono necessitare di avere vettori con dimensioni variabili, per fare ciò basta memorizzare il vettore in un array pari alla dimensione massima del vettore, per fare ciò però potrà capitare che in **determinati momenti non tutte le celle saranno utilizzate**.

Per gestire l'occupazione parziale ci sono due soluzioni: la prima è memorizzare il numero degli elementi validi in una ulteriore variabile, gli elementi successivi al secondo **non hanno nessuna importanza** ; Altra

soluzione è identificare il primo elemento non utilizzato, tale elemento di chiama **terminatore** (ad es si può utilizzare 0 per valori non nulli o -1 per valori positivi).

Le due implementazioni di vett dinamico sono esempi di implementazione di oggetto astratto, il vettore dinamico, mediante uno o più oggetti concreti. Processo di astrazione: l'oggetto astratto astrae dai dettagli su com'è implementato, sia che sia implementato con un array più un contatore, sia che lo sia con solo un array, ma ha comunque le stesse identiche caratteristiche.

Mediante un array si può definire un oggetto astratto vettore, di cui si realizzano: lunghezza variabile mediante contatore numero di elementi validi o mediante elemento terminatore e assegnamento mediante una funzione in cui si assegnano gli elementi uno ad uno.

Esiste un oggetto astratto di tipo vettore (chiamato vector) che fornisce già queste e molte altre operazioni (non lo useremo in questo corso).

Sappiamo già che la memoria di un programma è utilizzata per memorizzare gli oggetti di questo, ma non solo, contiene anche info non manipolabili da programmatore, come il codice delle funzioni (tradotto in LM) o codice e strutture dati aggiuntive di supporto al programma stesso.

Quando compiliamo, non traduciamo solo il programma in linguaggio macchina, il compilatore aggiunge ulteriore codice che effettua operazioni di supporto all'esecuzione del programma (es: inizializza i par. formali con i par . attuali, alloca spazio in memoria quando viene eseguita la definizione di una variabile/cost, ecc…). per questo accedere al di fuori delle celle di memoria dedicate ad oggetti correttamente definiti è un **errore di gestione della memoria** , se l'accesso avviene in lettura si leggono di fatto errori casuali, se l'accesso avviene in scrittura si rischia la cosiddetta corruzione della memoria del programma, ossia la sovrascrittura del contenuto di altri oggetti definiti nel programma, se si corrompono le strutture dati aggiuntive il comportamento del programma diventa **impredicibile**.

I processori moderni permettono di suddividere la memoria in porzioni distinte e stabilire quali operazioni si possono o no fare in certe porzioni, se un programma tenta di scrivere o leggere in una porzione in cui non ha il diritto di farlo, viene generata dal processore una eccezione hardware di accesso illegale alla memoria = sono un meccanismo del processore che fa interrompere l'esecuzione dell'istruz in corso e saltare immediatamente all'esecuz di codice speciale dedicato alla gestione dell'esecuz.

Tipicamente il codice di gestione delle eccezioni termina immediatamente il processo.

In UNIX l'eccezione scatenata da un accesso illegale alla memoria è tipicamente chiamata **segmentation fault**, e il codice di gestione del Segmentation termina forzatamente il processo.

In conclusione accedere fuori da un array è un errore: logico (non ha senso accedere ad un elemento al di fuori dell'array stesso) e di gestione della memoria (corruzione della memoria in caso di accesso in scrittura).

Nuova forma costrutto for

Dato un array A di elementi di un qualche tipo T , si può scrivere for ( [const] T &<identificatore>: A) <istruzione_composta> con identificatori e istruzioni qualsiasi.

L'identificatore è visibile nell'istruzione composta che costituisce il corpo del **for** e il ciclo è eseguito tante volte quanti sono gli elementi dell'array; in generale all'i-esima iterazione il riferimento si riferisce all'i-esimo elemento dell'array, permette quindi di leggere il valore di tale elemento o di aggiornare il valore di tale elemento → vantaggi: estrema semplicità =nessun'inizializzazione, nessuna condizione e nessun indice da gestire. Svantaggi= non si compila senza nuovo standard.

Lezione 14 Input stream

Stream : sequenza di caratteri

Istream: meccanismo per convertire sequenze di caratteri in valori di vari tipi

Standard istream: **cin** tipicamente associato al terminale da cui è fatto partire il programma, appartiene al namespace std ; l'input stream è riempito in qualche modo dal sistema in cui gira il programma.

I valori si leggono con l'operatore >> (leggi, estrai ), se la lettura ha successo allora i caratteri letti per decidere il valore da immettere sono eliminati dallo stream

Input formattato: le cifre sono convertite in numeri se il tipo delle variabili in cui si scrive è intero o reale (spazi bianchi saltati se non si usa manipolatore); ciascun stream ha un proprio stato (insieme di flag, valori booleani).

Errori e condizioni non standard sono gestiti assegnando o controllando in modo appropriato lo stato, una operazione che fallisce porta lo stream in stato di errore.

Classica condizione che causa il fallimento e porta uno istream in stato di erore : leggere la marca EOF

La si incontra solo se si è raggiunta la fine del file, nel caso UNIX Incontriamo EOF solo se l'utente preme ctrl+d su una riga vuota.

Una operazione di input che fallisce è una vera e propria operazione nulla: nessun carattere è rimosso dallo stream di input, prima standard C++11 la variabile restava inalterata, con c++1 si immette 0 nella variabile di destinazione.

Cin e Cin>> sono **espressioni**.

Le due operazioni si possono usare dove è atteso un valore booleano e in tal caso il significato del loro valore è : vero se l'operazione può aver successo perché lo stream è in stato BUONO; Falso se lo stream è in stato non-buono.

**Funzioni membro** : per invocarle bisogna utilizzare la notazione a punto.

Una volta in stato non buono, lo cin ci rimane finchè i flag non sono resettati (le operazioni in stato non-buono sono operazioni nulle; istruzione per resettare lo stato dello stream cin.clear();

si può controllare se si è raggiunto l'EOF con la funzione membro eof(), ritorna true se si è raggiunto l'eof.

Ostream = meccanismo per convertire valori di vario tipo in sequenza di caratteri, output formattato : operatore <<, standard output e standard error: cout e cerr.

La shell prima di far partire un programma, aggancia lo stdout ad un oggetto speciale del sistema operativo, tramite quale il terminale legge i caratteri che deve far apparire, in particolare il terminale è sempre in stato **bloccato** in cui aspetta uno dei due seguenti eventi: 1 segnalazione da parte di questo oggetto del fatto che è arrivato un nuovo carattere da far apparire sullo schermo "virtuale" del terminale emulato
2 la segnalazione che è stato premuto un tasto sulla tastiera del terminale emulato; quando succede uno di questi due eventi, il terminale si sveglia, fa quello che deve fare e si blocca di nuovo in attesa del prossimo evento.

Quando scriviamo un numero (97) sullo stdout appare il carattere ('a') perché l'editor del programma, il compilatore ed il terminale **utilizzano tutti la stessa codifica per il carattere**.

Anche il flusso di caratteri è una sequenza di 8 bit ciascuna, se immettiamo sullo stdout cout<<"Ciao"<<endl; la sequenza costituisce una stringa seguita dal carattere newline.

In particolare all'esecuzione dell'istruzione, sullo stdout viene immesso 'c' 'i' 'a' 'o' '\n' nel caso venga usata la tabella ASCII, numericamente si avrebbe 67 105 97 111 10.

Supponiamo che il cout sia configurato per la stampa dei numeri in notazione decimale, l'istruzione cout<<12; manda il numero 12, ovvero la sequenza di caratteri che rappresentano le cifre del numero 12 in base 10, ossia '1' '2' in termini di sequenza di numeri su stdout, che non è uguale a 1 2 , ma, usando ad esempio la codifica ASCII è uguale a 49 50, ed in bit è rappresentata dalla sequenza 00110001 00110010 (non è la stessa sequenza di bit utilizzata per rappresentare il numero 12 mediante un oggetto di tipo **int** in memoria perché in base 2 il numero 12 sarebbe 1100 e gli int occupano 4 byte sulle macchine attuali).

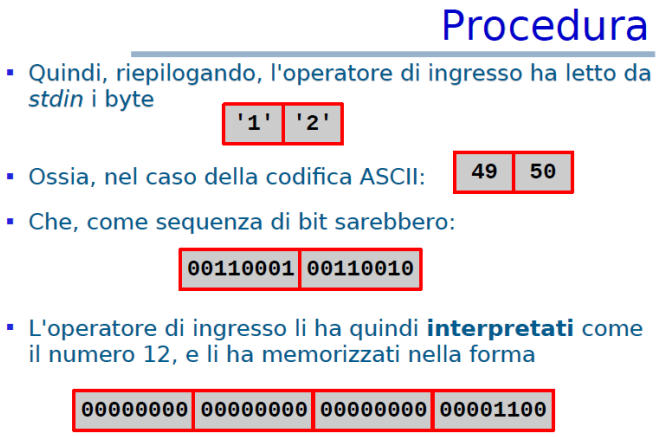
Quando si legge un carattere da stdin ci sono due possibilità: se sullo stdin ci sono già dei caratteri, si consuma il primo della sequenza e si mette esattamente il suo valore all'interno della variabile; se sullo stdin non ci sono già caratteri, il programma si blocca in attesa che finalmente vi arrivino.

I caratteri immessi finiscono sullo stdin perché la shell prima di far partire il programma aggancia lo stdin ad un oggetto speciale, sul quale il terminale spedisce i caratteri che vengono immessi da tastiera, quindi quando il programma legge un carattere da stdin, consuma il carattere in testa alla sequenza dei caratteri immessi, tale carattere **viene rimosso** dall'oggetto ed il prossimo da leggere sarà quello che lo seguiva (se presente).

In forma canonica il terminale prevede la pressione del tasto "invio" per l'invio dei caratteri.

Nota: il programma si blocca solo se lo stdin è vuoto, quando viene eseguita l'istruzione di lettura di un carattere da cin, altrimenti legge il primo carattere disponibile senza bloccarsi.

Esempio pratico: se si esegue l'istruzione cin>>a, e sullo stdin ci sono i caratteri 'c' 'i' 'a' 'o' '\n', dentro ad a finisce il codice del carattere C, ossia in codice ASCII il numero 67, e dallo stdin viene rimosso il primo byte per cui rimane 'i' 'a' 'o' '\n', una successiva lettura, leggerebbe il carattere i, rimanendo quindi 'a' 'o' '\n' e così via… stessa cosa per i numeri se variabile è tipo int, dopo aver letto un numero, questo finisce nella variabile int (in memoria) sotto forma di rappresentazione binaria.




Caso di fallimento: tipo int nello stdin vengono immessi ' ' 'z' '\n', l'operatore salta (consumandolo) lo spazio, dopo trova la z e l'interpretazione fallisce, per cui la variabile n rimane inalterata (o prende 0 in C++11) e sullo stdin rimane 'z' '\n'.

A causa del fallimento il cin andrebbe in stato di errore, per effettuare nuove letture bisognerebbe usare clear(), però la lettura fallirebbe di nuovo e sull'istream rimarrebbe ancora la solita sequenza, in sostanza sarebbe comunque impossibile riuscire a leggere tali cifre, in testa allo stdin continua a rimanere un carattere che fa fallire la lettura di un intero.

Si possono rimuovere incondizionatamente caratteri da un istream con la funzione <istream>.ignore() = ignora, ossia rimuove il prossimo carattere dell'istream.

Si può svuotare l'istream fino al prossimo newline con la funzione ignore se si una la seguente sintassi:

```
<istream>.ignore(std::numeric_limits
                 <std::streamsize>::max(),
                 '\n') ;
```

(attenzione: l'istream non deve essere già in stato di errore, e potrebbe farcelo andare lei per EOF)

Bufferizzazione

Immaginiamo chee l'oggetto cout memorizzi temporaneamente in un proprio array di caratteri nascosto i caratteri e che li scriva effettivamente su stdout solo quando arriva a contenere una riga, sarebbe molto più efficiente di un singolo carattere per volta.

Tale array è un esempio di **buffer** (memoria temporanea) = è un array temporaneo di byte utilizzato nelle operazioni di I/O, ci si appoggia i dati prima di trasferirle nella destinazione finale (è molto efficiente).

Le operazioni di uscita con gli stream sono tipicamente bufferizzate, (byte nascosti in un buffer nascosto) vengono spediti tutti i caratteri assieme quando si inserisce il newline, eventuali caratteri ancora presenti nel buffer vengono automaticamente spediti sullo stdout al termine del programma (se termina in modo naturale).

La dimensione del buffer nascosto è statica, se si prova ad inserire ulteriori byte quando il buffer è pieno, anche in questo caso viene svuotato per fare posto ai nuovi byte; ultimo caso, il buffer si svuota anche prima di una lettura da stdin, l'utente deve avere le informazioni in uscita dal programma prima di inserire a sua volta nuove info.

Problemi di **incoerenza delle informazioni** se ad esempio un programma è terminato forzatamente dopo una scrittura su cout senza il new line, i corrispettivi caratteri potrebbero non essere mai passati allo stdout.

Si può scatenare lo svuotamento del buffer con l'endl, che aggiunge il newline, altrimenti si può svuotare anche con il manipolatore flush.

Creazione nuovi stream, bisogna specificare l'oggetto a cui è associato, un tipico oggetto a cui associare uno stream è un **file.**

I seguenti tipi di stream sono da associare ai file e sono supportati direttamente dalla libreria standard del C++ : ifstream: file stream di ingresso (lettura), ofstream file stream di uscita (scrittura), fstream : file stream di ingresso/uscita; presentati in <fstream> (tutti e tre assieme).

File= una sequenza di caratteri (byte) che, come vedremo, potrà essere letta attraverso un ifstream (o fstream opportunamente inizializzato), o modificata attraverso ofstream (o di nuovo fstream).

Un (i/o)fstream viene associato ad un file mediante un'operazione chiamata **apertura** del file, da li in poi tutte le operazioni di i/o fatte sullo stream si tradurranno in identiche operazioni sul contenuto del file; il sistema operativo si occuperà di tutti i dettagli necessari per eseguire le operazioni sulla macchina reale.

Come nome del file si può indicare sia un percorso relativo, sia uno assoluto

Es assoluto: /home/giovanni/dati.txt (file di nome dati.txt nella cartella /home/giovanni)

Es relativo 1 : giovanni/dati.txt (file dati.txt nella sottocartella paolo della cartella corrente)

2 dati.txt (file di nome dati.txt nella cartella corrente).

Un programma può aprire più di un file, ma l'apertura può fallire in vari modi (es tentativo apertura di file inesistente), è opportuno controllare sempre l'esito dell'operazione di apertura prima di utilizzare un (i/o) fstream.

Un file è aperto in input definendo un oggetto di tipo ifstream e passando il nome del file come argomento:

```
ifstream f("nome_file") ;
        if(!f) cerr<<"L'apertura è fallita\n";
```

un file è aperto in output definendo un oggetto di tipo ofstream e passando il nome del file come argomento : `ofstream f("nome_file")` (+ if di controllo del fallimento)

Se non esiste un file aperto in scrittura viene **crearto,** altrimenti viene **troncato a lunghezza 0**; il contenuto precedente è perso.

Un file può essere aperto per l'ingresso e/o l'uscita definendo un oggetto di tipo fstream e passando il nome del file come argomento, deve essere fornito un secondo argomento openmode (ios_base::in oppure ios_base::out).

Per gli fstream si possono usare tutti gli operatori di stato e le funzioni di utilità per la formattazione viste per gli stream di i/o standard.

Si può controllare quindi lo stato d un oggetto usando il suo identificatore in una espressione condizionale.

Si possono realizzare letture/scritture formattate con la stessa sintassi di un i/o stream.

L'accesso al file avviene in modalità **sequenziale**= ogni lettura/scrittura viene effettuata a partire dal byte dello fstream successivo all'ultimo byte su cui ha lavorato l'operazione precendete.

(si può aprire anche con la funzione open, che non vedremo in queste lezioni).

Per gli stessi motivi (efficienza) dello stdout, anche le op di uscita su ofstream sono bufferizzate (senza endl non è garantita l'operazione di scrittura effettuata immediatamente).

Un file può essere chiuso invocando la funzione close(); all'atto della chiusura è garantito lo svuotamento del buffer e l'aggiornamento del file.

Il file associato è chiuso automaticamente (e aggiornato) quando finisce il tempo di vita dell'fstream (per esempio quando termina la funzione in cui l'oggetto è stato definito o al termine dell'intero programma, ma non per chiusura forzata del SO).

Se si vuole aprire un file esistente in scrittura senza troncarlo a lunghezza 0, bisogna aprirlo nella modalità append, lo si fa facendo passare ios_base::app come secondo parametro all'atto della definizione dell'ofstream; i byte inseriti verranno aggiunti a partire dal fondo.

Un oggetto di tipo stream può essere passato per riferimento ad una funzione, il tipo di un parametro formale attraverso il quale passare un istream o un ostream per riferimento è ovviamente istream & oppure ostream &.

Cin può essere passato come parametro attuale in corrispondenza di un parametro formale di tipo : istream &

Cout può essere passato come parametro attuale in corrispondenza di un parametro formale di tipo ostream &

Gli (i/o) fstream possono essere passati per riferimento dove sono attesi gli (i/o) stream

Un ifstream può essere passato coem par. attuale in corr di un par formale di tipo istream &

Un ofstream può essere passato coem par. attuale in corr di un par formale di tipo ostream &

Ciò permette di scrivere funzioni che con lo stesso codice possono operare indifferentemente su cin/cout o su file.

Lezione 15

Stringa= sequenza di caratteri (definizione di un tipo di dato astratto)

Letterale stringa (costante senza nome) = sequenza di caratteri che costituisce la stringa, delimitata da

doppi apici (esempio: la costante letterale per la stringa *sono una stringa* è "**sono una stringa**"

In C/C++ non esiste propriamente il tipo stringa, è implementato concretamente mediante un array di caratteri terminati da un carattere **terminatore**, il terminatore è il carattere **'\0',** come per i vettori è disponibile anche un tipo astratto stringa (string) con interfaccia di più alto livello di un array di caratteri (non in questo corso).
Sintassi : [const] **char** <identificatore> [ espr-costante>]; esempio: char stringa[6] (uno spazio va utilizzato per il terminatore, quindi la stringa ha 5 caratteri.

Char stringa [n] allora spazio per N-1 caratteri, ma è possibile memorizzare anche una stringa di dimensione inferiore, e le celle dopo il terminatore sono **concettualmente vuote** (contengono un valore che però non viene preso in considerazione
Una stringa p implementata mediante un array di caratteri, ma un array di carattere non è necessariamente una stringa (per essere una stringa necessita il terminatore.

Ci sono 3 modi per inizializzare una stringa: 1) char nome [6] = {'m','a','r','c','o'} 2 char nome [6]="marco" 3 char nome[]= "marco" (in terminatore viene inserito dal compilatore).

Se non si tratta di una inizializzazione, l'unico modo per assegnare un valore ad una stringa è carattere per carattere (compreso di inserimento del terminatore)

Se una stringa viene passata al cout con l'operatore <<, vengono stampati tutti i caratteri dell'array finchè non si incontra il terminatore mentre se la stringa serve per memorizzavi quello che si legge da cin con l'operatore >>, vi finisce dentro solo la prossima **parola**, ovvero sequenza di caratteri non separati da spazi ciò che è dopo lo spazio rimane sul terminale.

Errori tipici : definire un array e inizializzarlo successivamente, copiare una stringa in un'altra con l'assegnamento.

Differenze tra ‘A’ ed “A”: la prima è rappresentabile in un oggetto di tipo char : char c=’A’ ; il secondo è rappresentabile in un array di due caratteri : char s[2] = “A”;

Stringhe statiche = dimensione fissa; stringhe dinamiche =dimensione massima definita a tempo di scrittura

Per le stringhe valgono la stessa sintassi e semantica del passaggio alle funzioni degli array, sono quindi passate sempre per riferimento ed è opportuno utilizzare il const se non serve modificare la funzione.

Funzioni di libreria per la gestione delle stringhe, presentata in <cstring>:

- Strcpy (stringa1, stringa2) = copia il contenuto di str2 in str1 (sovrascrive)
- Strncpy (str1, str2 ) = copia i primi n caratteri di str2 in str1
- Strcat (str1, str2) = concatena il contenuto di stringa2 in stringa1
- Strcmp (str1, str2) = confronta stringa2 con stringa1: 0 (uguali), >0 (stringa1 è maggiore di stringa 2), <0 (viceversa)

STRUCT

Enupla ordinata di elementi detti **membi, o campi**, ciascuno dei quali ha un suo nome ed un suo tipo (in altri linguaggi è spesso chiamato **record**), il tipo struttura differisce da un array perché gli elementi non sono vincolati dall’essere tutti dello stesso tipo e ciascun elemento ha il suo nome.

Mediante il costrutto **struct** si possono di fatto dichiarare nuovi tipi di dato (esempio: tipo di dato persona)

Dichiarazione: struct <nome_tipo> {<lista_dichiarazioni_campi>} ;

definizione di oggetti di un tipo strutturato <nome_tipo>: [const] <nome_tipo> <identificatore1, identificatore 2>,….;

Si possono definire oggetti anche all’atto della dichiarazione del tipo stesso.

Per selezionare i campi di un oggetto strutturato si utilizza notazione a punto: <nome_oggetto>.<nome_campo>

Uno struct può essere inizializzato: elencando i valori dei campi fra parentesi graffe o copiando il contenuto di un altro oggetto dello stesso tipo; l’assegnamento equivale alla copia campo per campo (i due oggetti devono essere dello stesso tipo struttura, non è consentito farlo con nomi di tipi diversi).

Per ottenere la copia di due array senza usare un ciclo basta definire semplicemente un tipo struttura che contiene semplicemente un array, e si può definire un array di oggetti di tipo struct.

Per passare un array di oggetti di tipo struct si usa la stessa sintassi del passaggio di array di oggetto di tipo primitivo.

Quando c’è da scegliere tra struct e un array, (esempio trapezio) la scelta ricade sugli struct per 1 maggiore leggibilità (i campi sono acceduti mediante nomi significati ) e c’è una 2 migliore organizzazione dei dati (info logicamente correlate, sono raggruppate assieme nello stesso tipo di dato); 3 maggiore leggibilità delle chiamate di funzione, un errore che bisogna evitare è quello di invocare qualche funzione passando troppi parametri ( il numero oltre il quale il lettore si perde è **7**), spesso si sbaglia a passare uno ad uno i campi interessati, quando in questi casi conviene passare l’intero oggetto.

Se la funzione lavora su alcuni campi, conviene passare solo quei campi (se la funzione continua ad avere troppi parametri allora l’errore può essere una funzione troppo lunga e complessa.

Gli oggetti struct possono essere passati/ritornati per valore, nel parametro formale finisce la copia campo per campo del parametro attuale, quindi sarebbero copiati tutti gli elementi di eventuali array, può essere troppo costoso, quindi si può passare per riferimento (se non si vuole la modifica con il const).

MATRICE

Insieme di elementi distribuiti in n righe e m colonne.

Sintassi definizione di una variabile o di una costante **matrice bidimensionale statica :** [const] <tipo_elementi_matrice> <identificatore> [<espr-costante>] [espr-costante] ;

esempio di matrice 4x3 di tipo double: double mat[4][3];

Sintassi della definizione di una variabile con nome di tipo matrice statica k-dimensionale: [const] <tipo_elementi_matrice> <identificatore> [<espr-cost_1>] [<espr-cost_2>] …. [<espr-cost_k>];

per accedere ad un elemento bisogna fornire tanti indici quante sono le dimensioni.

Inizializzazione: int mat[3][4] = { {2, 4, 1, 3},
{5, 3, 4, 7},
{2, 2, 1, 1} } ;

Il numero di colonne **deve** essere specificato

Il numero di righe può essere omesso, nel qual caso coincide col numero di righe che si inizializzano

Elementi non inizializzati hanno valori casuali o nulli (cas se sono locali, nulli se globali).

Non si possono inizializzare più elementi di quelli presenti.

Sostanzialmente una matrice è un array di array, es: int mat [M][N] definisce un array di M array da N elementi ciascuno, ossia M righe da N colonne ciascuna, in memoria siccome un array è una sequenza contigua di elementi, allora un array di array è una sequenza contigua di array in memoria

Anche gli array di array sono **passati per riferimento.**

la dichiarazione di un parametro formale è la seguente :

[const] <tipo_elementi> <identificatore> [] [numero_colonne] → nessuna indicazione del numero di righe!!

La funzione pertanto non conosce implicitamente il numero di righe della matrice; nell'invocazione della funzione, una matrice si passa **scrivendone semplicemente il nome**.

Per analogia, un array di stringhe si realizza mediante una matrice di tipo **char**.

Lezione 16

La dimensione di tutti gli oggetti visti fin ora deve essere definita a tempo di scrittura del programma, può essere però che non si sappia a priori, per poter allocare in memoria oggetti le ui **dimensioni sono determinate durante l'esecuzione del programma**, si applica l'allocazione dinamica della memoria.

Prima dell'inizio di un processo, il S.O. riserva al processo spazio di memoria di dimensioni predefinite: locazioni consecutive di memoria (tipicamente di un byte l'una) questo spazio è organizzato in segmenti distinti, uno di questi è chiamato **memoria libera o dinamica** oppure **heap**, è possibile allocare oggetti di **dimensione arbitraria** in memoria dinamica in **momenti arbitrari** fino ad esaurimento di questa.

Gli oggetti allocati in memoria dinamica sono detti **dinamici**, ora analizzeremo solo gli array dinamici, array allocati durante l'esecuzione del programma.

Un array dinamico può essere allocato mediante l'operatore **new**: new<nome_tipo>[<numero_elementi>];

allora un array di <num_elementi> oggetti di tipo <nome_tipo>, non inizializzati (valori casuali) <num_elementi> può essere un'**espressione aritmetica qualsiasi.**

Il new può essere utilizzato per allocare oggetti dinamici di ogni tipo, ma per ora vedremo solo array. Gli elementi dell'array hanno **valori casuali**, gli oggetti dinamici sono **oggetti senza nome** ai quali si accede con il loro indirizzo.

L'operatore **new** ritorna proprio **l'indirizzo dell'oggetto allocato**, possiamo accedere a tale indirizzo, ma prima bisogna memorizzarlo da qualche parte, ci serve quindi un oggetto di tipo **puntatore**.

Un oggetto di tipo puntatore ha per valore un indirizzo di memoria (che è un numero naturale) la definizione di un puntatore è: [const] <tipo_oggetto_puntato> * [const] <identificatore> [= <indirizzo>];

il primo const ci va se si punta ad un oggetto immodificabile, il secondo ci va se il valore del puntatore dopo essere stato inizializzato non può essere modificato.

Accesso agli elementi di un array dinamico è possibile in modo **identico** agli array statici, selezione con indice mediante parentesi quadre (gli indici partono sempre da 0).

Un oggetto di tipo puntatore ha per valori un sottoinsieme dei numeri naturali, nel caso contenga 0, viene detto **puntatore nullo**; il riferimento ad un puntatore segue le **stesse regole di visibilità di qualsiasi altro identificatore. Ma non confondiamo un puntatore con l'array a cui punta!!** Il puntatore serve per mettere da parte l'indirizzo dell'array e accedere all'array stesso; una volta allocato un array esiste fino alla fine del programma (a meno che non venga deallocato) e continua ad esistere anche se non esiste più il puntatore che contiene il suo indirizzo.

SI può deallocare un array dinamico (liberare lo spazio in memoria da esso occupato) con l'operatore: delete[] <indirizzo_oggetto_da_deallocare> ; può essere applicato solo all'indirizzo di un array dinamico allocato con l'operatore **new**, altrimenti errore gestione della memoria.

Passaggio di un array dinamico ad una funzione:

[const] <nome_tipo> <identificatore> [] oppure [const] <nome_tipo> * <identificatore> entrambe le sintassi sono perfettamente equivalenti e si possono usare sia per array statici che per dinamici; le dimensioni dell'assay passato non sono implicitamente note alla funzione chiamata, il passaggio dell'array è **per riferimento**.

A livello di linguaggio, il passagio di un array non è per riferimento, ma per valore, il parametro formale contiene infatti una copia dell'indirizzo dell'array, siccome il parametro formale contiene l'indirizzo dell'array, tramite il par formale si accede esattamente all'array il cui indirizzo è passato come parametro attuale, quindi ogni modifica effettuata all'array puntato dal parametro formale si riflette sull'array di cui si è passato l'indirizzo.

Scrivere il nome di un array statico come parametro attuale in corrispondenza di un parametro formale di tipo puntatore equivale a scrivere l'indirizzo del primo elemento dell'array. Una funzione può ritornare l'indirizzo di un array dinamico, il tipo di ritorno deve essere: [const] <nome_tipo>*

Come sappiamo, il riferimento è un tipo derivato, dato un tipo di partenza, si può definire un riferimento a tale tipo, se il tipo di partenza è un puntatore, allora un riferimento ad un oggetto di tipo puntatore si definisce come segue: [const]<nome_tipo>*&<identificatore>;

Più puntatori possono puntare allo stesso oggetto, possono quindi esserci effetti collaterali; i puntatori in quanto variabili, possono essere utilizzati anche senza inizializzazione, causando errori logici e di accesso della memoria, possono essere inoltre riassegnati in ogni momento, queste caratteristiche dei puntatori possono portare ad errori più difficili da trovare:

**dangling reference (pending pointer) =** quando un puntatore punta ad una locazione di memoria che non contiene alcun oggetto allocato, accade perché il puntatore non è stato inizializzato o perché l'oggetto è stato deallocato, se si usa il pending pointer possono sorgere errori che portano ad un comportamento impredicibile del programma.

In assenza di memoria libera disponibile, il New fallisce, viene generata una eccezione, se non gestita il programma viene terminato forzatamente; se si vuole, si può gestire l'eccezione, oppure agganciare il fallimento dell'operatore ad una propria funzione (sarà invocata in caso di fallimento)

**Memory leak:** esaurimento inaspettato della memoria, causato da mancata deallocazione di oggetti dinamici non più utilizzati (spesso causato dalla perdita dell'indirizzo).

È possibile allocare array dinamici di oggetti di qualsiasi tipo: stringa dinamica, array di struct e array di array.

Es: stringa di dimensioni definite da stdin:

```
        int dim;
do cin>>dim; while (dim<= 0) ;
        char * const str = new char[dim+1];
```

es: array dinamici di struct :

```
        struct persona
{
        Char nome_cognome[41];
        char codice_fiscale[17];
        float stipendio;};
main()
{
        Int dim;
        do cin>>dim; while (dim<=0);
        persona * const t = new persona[dim];
```

Matrici dinamiche, la matrice è un array di array, quindi una matrice dinamica è un array dinamico di un array, ogni elemento dell'array dinamico è a sua volta un array , ci concentriamo solo sul caso in cui sia un array statico: int (*p) [10] = new int[n][10];

deallocazione: delete [] p;

Per passare una matrice dinamica bidimensionale occorre un parametro della forma: [const] <nome_tipo> (*<identificatore>) [<espr_costante>].

Nel caso si ometta il nome del parametro in una dichiarazione, la sintassi diviene: void fun(int (*)[10]);

Si accede agli elementi di una matrice dinamica utilizzando la stessa sintassi che si utilizza per una matrice statica.

Lezione 17

Gli operatori di ingresso/uscita visti finora, ossia >> e <<, interpretano il contenuto di uno stream come una sequenza di caratteri, le operazioni di i/o in cui uno stream è visto come sequenza di caratteri di definiscono operazioni **formattate**.

Esistono anche operazioni i/o **non formattate**, vedono lo stream dal punto di vista di più basso livello, ossia come una mera sequenza di byte, non effettuano trasformazione di alcun tipo, trasferiscono semplicemente sequenze di byte da uno stream alla memoria del processo o viceversa.

Così come per le formattate, anche le lett/scrit non formattate accedono agli stream in modo sequenziale, ogni lett/scrit viene effettuata a partire dal byte dello stream successivo all'ultimo byte dello stream su cui ha lavorato l'operazione precedente.

Le funzioni di lettura e scrittura non formattate che vedremo sono chiamate **funzioni membro** degli stream; per invocarle bisogna utilizzare la notazione a punto.

Come già visto, un **buffer** è un array di byte utilizzato nelle operazioni di i/o, il tipo char ha esattamente la dimensione di un byte, quindi il tipo **char** è utilizzato anche per memorizzare byte nelle letture e scritture non formattate, gli istream dispongono delle seguenti funzioni membro per input non formattato:

- Get ()
- Get (char &)
- Get (char *buffer, int n, [char delimitatore])
- Read ( char *buffer, int n)
- Gcount ()

Get()= ritorna su int il valore del prossimo byte nello stream di ingresso a cui è applicata (ritorna EOF se sei ad EOF)

Get(char &c)= preleva un byte e lo assegna alla variabile passata come argomento, la variabile è lasciata inalterata in caso di fine input, possiamo dire che tale funzione ritorna lo istream a cui è applicata, possiamo usarlo come istream in espressioni che si aspettano un booleano

Get(char *buffer, int n, [char delimitatore])= legge una sequenza di byte dallo stream di ingresso e li copia nell'array buffer, aggiungendo automaticamente il terminatore, la lettura va avanti finchè non si sono letti n byte oppure non si è incontrato il delimitatore (se omesso è '\n') (se tenta di leggere una riga vuota, mando l'istream in stato di errore).

Esempio evoluto di lettura di una riga: utilizzando la funzione get insieme alla funzione membro peek() (che ritorna il valore del prossimo byte presente su un istream, senza rimuovere tale byte) si può leggere da istream una riga alla volta.

Read( char *buffer, int n)= legge n byte e li copia nell'array **buffer** , non è previsto alcun delimitatore, né aggiunto alcun terminatore (è di più basso livello rispetto alla get).

Gcount()= ritorna il numero di byte letti nell'ultima operazione di lettura.

Put(char c)= trasferisce un byte (il contenuto di c) sullo stream di uscita

Write (const char *buffer, int n)= trasferisce i primi **n** byte dell'array **buffer** sullo stram di uscita (non è aggiunto alcun terminatore).

**Modo testo=** se si effettuano solo letture e scritture formattate su uno stream

**File di testo =** un file i cui byte sono da interpretarsi come codici di caratteri

Altrimenti si usa tipicamente la denominazione **file binario**.

Per lavorare su file binari sono molto comode le lett/scritt non formattate, perché permettono di ragionare in termini di pure sequenze di byte.

Dalla decima esercitazione: le informazioni sono memorizzate mediante rappresentazioni numeriche, e un file è semplicemente una sequenza di numeri, a cui è associato un nome; anche un file di testo è una sequenza di numeri, il quale numero rappresenta il codice ASCII di un carattere.

I file .bmp contengono immagini sotto forma di sequenze di bit, nel caso di immagine in bianco e nero, un bit a 1 può rappresentare un pixel nero, viceversa per lo 0.

Il suffisso di un file ci da solo un suggerimento su come vanno interpretati i byte contenuti nel file, l'effettivo tipo del file dipende dal modo in cui si deve effettivamente interpretare la sequenza di byte in esso contenuta.

Comandi per visualizzazione di un file:

- Cat
- More, less (per uscre: 'q')

Comadno per visualizzatore esadecimale: hd, dump, hexdump

Es: hd nomefile, se mostra solo numeri, aggiungete l'opzione -C prima del nome del file.

Dalla lezione

Possiamo passare un array come argomento attuale nella posizione corrispondente ad un parametro formale di tipo <tipo> * perché passare il nome di un array è equivalente a passare l'indirizzo del primo elemento dell'array.

Il tipo char * memorizza indirizzi, possiamo quindi scriverci dentro l'indirizzo di qualsiasi oggetto, possiamo in particolare, anche scriverci dentro l'indirizzo di oggetti di **tipo diverso** da array di caratteri.

Le funzioni di I/O non formattate, si aspettano però solo array di caratteri per usare tali funzioni dobbiamo perciò convertire il tipo dell'indirizzo di un oggetto diverso da un array di caratteri mediante: reinterpret_cast<char *>(<indirizzo>); (se necessario anche const).

Dato che una sequenza di byte non è altro che un **array di byte,** ma sappiamo che un byte può essere rappresentato esattamente su di char, quindi qualsiasi sequenza di byte può essere rappresentata con array di char.

Quindi mediante il reinterpret_cast, diciamo che reinterpreta l'indirizzo dell'oggetto come l'indirizzo a cui inizia una sequenza di byte, rimane il problema di sapere la lunghezza di tale sequenza di byte, più in generale potremmo voler trasferire una sequenza di byte relativa solo ad una porzione dell'intero oggetto.

Dato un generico array di elementi di qualche tipo, possiamo calcolare la lunghezza della sequenza di byte occupati da tutti o da una parte degli elementi dell'array utilizzando l'operatore **sizeof** per conoscere le dimensioni di ogni elemento e moltiplichiamo per il numero di elementi.

Le funz di lettura e scrittura non formattate, si possono usare anche per file non necessariamente solo binari, basta scrivere i byte che rappresentano i codice dei caratteri desiderati, un byte alla volta, o più alla volta facendo delle write di interi array di caratteri (come nell'esempio char).

Scrivere gli elementi uno alla volta in un file mediante un ciclo scritture non formattate è inefficiente, tipicamente si fa per scritture formattate, è efficiente perché l'operatore bufferizza le operazioni.

Come abbiamo visto, il nome di un array denota l'indirizzo di un array, pertanto passare un array come parametro attuale equivale a passarne l'indirizzo; ma per passare ad un funzione di i/o l'indirizzo di un oggetto diverso da un array (ad es un singolo intero o un singolo oggetto struttura) dovremmo utilizzare l'operatore indirizzo & (è un operatore unario prefisso ) sintassi: & <nome_oggetto>; ritorna l'indirizzo dell'oggetto passato per argomento (vedi esempio lezione 17 slide da 48 a 54).

Uno stream è **acceduto sequenzialmente** se ogni operazione interessa caselle dello stream consecutive a quelle dell'operazione precedente, si ha invece **accesso casuale** ad uno stream se per una operazione si sceglie arbitrariamente la posizione della prima casella coinvolta (per cin,cout,cerr è possibile solo l'accesso sequenziale).

La *casella* (ossia il byte) a partire dalla quale avverrà la prossima operazione è data da un *contatore*che parte da 0 (prima casella), il suo contenuto può essere modificato con le funzioni membro:

seekg(nuovo_valore) per file in ingresso (la g sta per get) oppure con argomento: seekg(offset, origine)

seekp(nuovo_valore) per file in uscita (la p sta per put) oppure con argomento: seekp(offset, origine)

l'origine può essere:

- Ios::beg = offset indica il numero di posizioni a partire dalla casella 0 (equivalente a non passare un secondo argomento)
- Ios::end = offset indica il numero di posizioni a partire dall'ultima casella (muovendosi all'indietro

Per gli ifstream è definita la funzione tellg() = ritorna il valore corrente del contatore

Per gli ofstream è definita la funzione tellp() = ritorna il valore corrente del contatore

Per concludere: i file sono strutture dati sono una delle strutture dati fondamentali per la soluzione di problemi reali, infatti nella maggior parte delle applicazioni i dati non si leggono (solamente) da input e non si stampano (solamente) su terminale, ma si leggono e si salvano su file; spesso si rende necessario gestire in modo efficiente grandi quantità di dati su supporti di memoria di massa

Lezione 18

Librerie: il solo insieme di istruzioni di un linguaggio di programmazione come il C/C++ **non** è sufficiente a scrivere un qualsiasi programma in grado per lo meno di interagire con l'utente; gli stessi oggetti cin e cout appartengono ad una libreria; libreria= raccolta di funzioni ed oggetti che permettono di effettuare determinati insiemi di operazioni.

Sia nel C che nel C++ è prevista una **libreria standard** = è sostanzialmente un sovrainsieme di quella del C, sia quella standard del C che quella del C++ sono costituite da moduli, ciascuno è praticamente una libreria a se stante, che fornisce funzioni ed oggetti per un determinato scopo, per utilzzare ciascun modulo è tipicamente necessario includere un ben determinato *header file.*

Per utilizzare i moduli della liberia standard C++ è bene includere degli header file il cui nome si ottiene a partire dal nome del corrispondente header file per il C, eliminando il suffisso .h ed aggiungendo una **c** all'inizio del nome, nel caso di C++ i nomi delle funz e degli oggetti di queste librerie sono definiti nello spazio dei nome std.

Diversamente dal C++, in C l'i/o formattato è realizzato mediante funzioni di libreria presente in <studio.h>, <cstdio> se volete utilizzare tali funzioni in C++, tra le principali : **printf (**output) e **scanf** (input).

Void printf (const char format[], …. ); …= → lista valori da stampare

La stringa **format** può contenere due tipi di oggetti: 1 caratteri ordinati (compresi quelli speciali) copiati tali e quali sullo stdout 2 Specifiche di conversione: utilizzate solo se sono passati ulteriori parametri, contenenti valori da stampare, dopo la stringa **format**; controllano l'interpretazione e quindi la traduzione in caratteri dei valori dei parametri aggiuntivi da stampare (bisogna inserire una specifica per ogni valore da stampare).

Nella posizione in cui appare una specifica di conversione nella stringa di formato , verrà stampato al suo posto il valore passato come ulteriore parametro, col formato determinato dalla specifica di conversione stessa (esempio di conversione: %<sequenza di caratteri che specificano tipo e formato da stampare>%)

Alcune specifiche più utilizzate sono: %d numero intero, da stampare in decimale; %g numero reale, da stampare in decimale; %c Carattere, tipicamente codifica ASCII; %s Stringa, tipicamente codifica ASCII.

La funzione printf ha numero di argomenti **variabile**, per ogni specifica di conversione si può aggiungere un parametro attuale contenente il valore da stampare.

Funzioni con un numero variabile di argomenti si definiscono **variadiche**; esistono sia in C che C++, per dichiararle si usa un'estensione della sintassi vista finora (non serve in questo corso).

Con << non dovevamo utilizzare specifiche di conversione perché determinava automaticamente il tipo e il formato dei valori, formato che si può comunque modificare attraverso manipolatori e funzioni membro.

Void scanf (const char format[], <indirizzo_variabile> ) ; indirizzo variabile in cui riversare ciò che si legge da stdin.

Forma semplificata in cui **format** può contenere solo una specifica di conversione (in generale anche **scanf** è invece una funzione variadica), tale specifica controlla l'interpretazione da dare ai caratteri letti da stdin per determinare il valore da memorizzare nella variabile passata come secondo argomento, il tipo si assume abbia la variabile.

Specifiche di conversione: %d interpretare i caratteri sullo stdin come cifre di un num intero, in decimale, da memorizzare in un **int**; %lg cifre di un numero reale, in decimale, da memorizzare in **double**; %c Carattere, tipicamente codifica ASCII, da memorizzare in un **char;** %s Stringa, tipicamente codifica ASCII, da memorizzare in un **char[].**

Se si sbaglia con le specifiche: errore logico ed errore di gestione della memoria, estremamente dannoso nella **scanf** (corruzione della memoria).

Con l'operatore >>, è necessario fornire specifiche di conversione perché l'operatore determina automaticamente il tipo dei valori, senza che sia necessario informarlo esplicitamente : eliminata la possibilità di sbagliare tipo o fornire un indirizzo errato! Dal tipo, l'operatore >> decide automaticamente anche l'interpretazione da dare ai caratteri su stdin, che si può comunque modificare attraverso manipolatori e funzioni membro.

Il controllo automatico degli operandi (<< o >>) fa si che il loro tempo di esecuzione non sia maggiore di quelli non formattati perché la scelta del codice da eseguire avviene a tempo di compilazione, si compila solo il codice appropriato.

Per l'i/o formattato su file in ci si usano due varianti di printf e scanf, chiamate **fprinf e fscanf** (non li vedremo in questo corso).

Direttiva #define, era il const in C, è una direttiva C/C++ per il preprocessore; comporta una sostituzione **testuale** del simbolo passato come primo argomento con qualsiasi sequenza di caratteri lo segua, prima della compilazione , nessuna dichiarazione/controllo di tipo, il simbolo sparisce **prima** della compilazione.

Anche in C ci sono struct e enum, ma a differenza del C++ la definizione di oggetti dei due tipi va fatta ripetendo ogni volta rispettivamente struct ed enum.

**Typedef:** sinonimi di tipi primitivi, oppure di tipi precedentemente dichiarati; aiutano tantissimo la leggibilità dei programmi, permettono di evitare dichiarazioni molto complesse e permettono di sostituire nomi di tipo di basso livello con nomi di tipo significativi nel dominio del problema; nelle applicazioni con problemi di overflow, o in generale in cui la conoscenza del tipo di dato a basso livello è importante, le dichiarazioni typedef possono essere dannose, perché non vedere direttamente il tipo di dato "concreto" nelle dichiarazioni può rendere le cose più complicate.

Allocazione array dinamici in C:

Mediante funzione di libreria **malloc** presentata in <stdlib.h> (<cstdlib> per C++) prende in ingresso la dimensione in byte dell'oggetto da allocare e ritorna l'indirizzo dell'oggetto, oppure 0 in caso di fallimento; esempio: <nome_tipo>*<identificatore>= malloc(<num_elementi>*sizeof(<nome_tipo>));

Deallocazione in C: mediante funzione di libreria **free:** presentata in <stdlib.h> (<cstdlib> per C++); prende in ingresso l'indirizzo dell'oggetto da deallocare, es: free(<indirizzo_array>);

In C non ci sono operatori per allocare/deallocare della memoria, ma come si è visto due funzioni di libreria, malloc opera ad un livello di astrazione più basso dell'operatore new, alloca una sequenza di byte lunga quanto le comunichiamo, al contrario, all'operatore new possiamo chiedere esplicitamente di allocare un array di un certo numero di elementi di un dato tipo.

Lezione 19

Proprietà di un algoritmo:

**Eseguibilità:** ogni azione deve essere eseguibile da parte dell'esecutore in un tempo finito

**Non- ambiguità**: ogni azione deve essere univocamente interpretabile dall'esecutore

**Terminazione :** il numero totale di azioni da eseguire, per ogni insieme di dati di ingresso, deve essere finito.

Inoltre, un algoritmo deve essere applicabile a qualsiasi insieme di dati appartenenti al dominio di definizione dell'algoritmo stesso; e deve essere costituito da operazioni appartenenti ad un determinato insieme di operazioni fondamentali, e operazioni non ambigue, ossia interpretabili in modo univoco qualunque sia l'esecutore (persona o macchina) che la legge.

**Efficienza :** misurata in numero di passi da effettuare in occupazione di memoria

**Determinismo:** esistono algoritmi volutamente non deterministici allo scopo di raggiungere una maggiore efficienza.

**Due algoritmi sono equivalenti quando:** hanno lo stesso dominio di ingresso e di uscita, e in corrispondenza degli stessi valori nel dominio di ingresso producono gli stessi valori nel dominio di uscita.

**Due algoritmi equivalenti:** forniscono lo stesso risultato, ma possono avere **diversa efficienza** (numero di passi da effettuare, quantità di memoria occupata) e possono essere profondamente diversi.

Al variare del linguaggio di programmazione, lo stesso algoritmo sarà codificato in un diverso programma.

**Costo computazionale:** efficienza in termini di tempo di esecuzione, il numero di passi che deve effettuare per ottenere l'obbiettivo per cui è stato definito.

Il numero esatto di passi elementari effettuati da un algoritmo per risolvere un problema può variare, in base all'insieme di valori di ingresso, magari fino ad una certa dimensione del problema c'è un certo costo, ma poi cambia; tipicamente il tempo di esecuzione di un algoritmo può diventare un problema pratico solo quando il numero di elementi su cui l'algoritmo lavora è molto grande; per risolvere questo problema bisognerebbe mettere un contatore per ogni passo che l'algoritmo esegue per ogni elemento.

**Pendenza =** crescita (esponenziale) dei passaggi

Ordine di costo= per dare la definizione vediamo un caso particolare: Dato il numero di elementi N su cui un algoritmo lavora (dimensione del problema), si dice che l'algoritmo ha **ordine O(N)** se esiste una costante c tale che il numero di operazioni elementari effettuate dall'algoritmo cresce con una pendenza non maggiore di c*N; definizione generale (semplificata) : dato il numero di elementi N su cui un algoritmo lavora (dimensione del problema), si dice che l'algoritmo ha costo di **ordine O(f(N))** se esiste una costante c tale che il numero di operazioni elementari effettuate dall'algoritmo cresce con una pendenza non maggiore di c*f(N).

Ordini di costo polinomiali:

- O(N): l'algoritmo effettua un numero di passi proporzionale al numero di elementi su cui si lavora
- $O(N^2)$: l'algoritmo effettua un numero di passi pari al quadrato del numeri di elementi su cui lavora
- $O(N^i)$: forma generale per l'ordine di costo polinomiale, per problemi grossi, ogni volta che i aumenta, il tempo di esecuzione diviene enormemente più lungo.

Ordine di costo esponenziale: $O(a^N)$: al crescere di N il tempo di esecuzione dell'algoritmo diviene così lungo da renderne impossibile il completamento in tempi ragionevoli.

Viceversa abbiamo ordine di costo costante: O(1): il numero di passi effettuati dall'algoritmo è **indipendente** dal numero di elementi su cui si lavora.

Ora disponiamo di strumenti efficaci per confrontare il costo computazionale (efficienza) degli algoritmi: l'ordine di costo permette di astrarre da dettagli che possono nascondere il vero stato delle cose (crescita esponenziale dei calcoli); ma se la dimensione di N è abbastanza piccola, allora potrebbe non valere la regola del costo computazionale.

Un algoritmo è utile se esiste un esecutore(spesso chiamato **automa esecutore)** in grado di eseguirlo, può essere istruito in modo efficace per eseguire un algoritmo se: 1) può essere programmato mediante un insieme di istruzioni (che è in grado di eseguire), le istruzioni sono sufficienti per eseguire i passi dell'algoritmo che si vuole far eseguire a tale esecutore e la sintassi e la semantica delle istruzioni sono complete e non ambigue.

**Linguaggio di programmazione:** insieme di istruzioni con le precedenti caratteristiche.

Quando programmiamo, presumiamo la presenza di un esecutore in grado di eseguire le istruzioni di un programma in C/C++.

**Macchina virtuale=** automa in grado di eseguire le istruzioni del linguaggio C/C++, supportare il concetto di variabile, costante con nome, tipo di dato, funzione. (nessuna macchina reale ha tali caratteristiche)

Macchina reale: Dal punto di vista reale, abbiamo eseguito i nostri programmi su degli elaboratori elettronici, ma sappiamo già che un processore può eseguire solo istruzioni macchina del processore stesso (**linguaggio macchina del processore=**sequenza di numeri).

Per rendere comprensibile ad un essere umano il linguaggio macchina, tipicamente si usa l'assembly= per ogni istruzione del linguaggio macchina, esiste una corrispondente istruzione nel linguaggio assembly, in assembly tale istruzione non è più individuata da un numero, ma da una stringa (tipicamente un nome abbreviato che ricorda lo scopo dell'istr. stessa).

**Registro=** una speciale cella di memoria (tipicamente > di 1 byte) interna al processore, scopo di tali registri è contenere gli operandi delle istruzioni aritmetiche, tali istruzioni tipicamente non possono utilizzare direttamente le celle di memoria principale dell'elaboratore come operandi, spesso è necessario copiare prima gli operandi all'interno dei registri.

Il linguaggio macchina è **il** linguaggio di **un** processore, è direttamente eseguibile senza intermediazione, processori differenti hanno linguaggi macchina differenti, quindi il linguaggio macchina **non è portabile**.

Infatti conviene codificare direttamente in un linguaggio macchina solo se necessitiamo di elevata efficienza e non cerchiamo di ottenere portabilità; la portabilità la otteniamo con un maggiore livello di astrazione.

I Linguaggi di alto livello si basano su un ipotetico elaboratore dotato di un processore le cui istruzioni non son quelle di un tipico processore reale, ma quelle del linguaggio stesso (ciò che abbiamo chiamato macchina virtuale); supportano **concetti ed astrazioni**, promuovono metodologie per agevolare lo sviluppo del software, hanno capacità espressive superiore rispetto al Lingu. Macc. Ed hanno centinaia di linguaggi di programmazione di alto livello di astrazione.

Quindi, in generale, un linguaggio di alto livello permette di codificare un algoritmo:

- Con costrutti e strutture dati più vicine al dominio del problema
- Non si cura dei dettagli di basso livello
- Mantiene un'ottima portabilità rispetto all'assembly.

Esistono tanti linguaggi perché si differenziano per ogni contesto applicativo (es scientifico : Fortran; Gestionale: Cobol; Sistemi operativi: C; applicazioni non di rete: C++,java; applicazioni di rete: Java)

Come abbiamo visto, non esistono processori il cui linguaggio macchina raggiunga un livello di astrazione confrontabile con quello di un linguaggio di programmazione di alto livello, quindi non esistono processori di eseguire più o meno direttamente programmi scritti in linguaggi di alto livello.

Per far si che un programma scritto in un qualunque linguaggio sia eseguibile da un elaboratore, bisogna **tradurlo** nel linguaggio di programmazione originario al linguaggio macchina del processore montato su tale elaboratore, operazione svolta da programmi chiamati **traduttori.**

I traduttori convertono il testo dei programmi scritti in un particolare linguaggio di programmazione (**programma sorgente)** nella corrispondente rappresentazione in linguaggio macchina (**programma eseguibile);** nel caso del **compilatore**, lo schema traduzione-esecuzione si percorre una volta sola e porta alla creazione del file eseguibile che sarà poi eseguito senza altri interventi; nel caso **dell'interprete** lo schema traduzione-esecuzione viene ripetuto tante volte quante sono le istruzioni del programma che saranno eseguito (ad ogni traduzione, segue l'esecuzione dell'istruzione stessa).

Differenze di linguaggi che tipicamente vengono interpretati e altri compilati:

- Interpretati : Basic, Javascript, Perl…
- Compilati: C/C++, Fortran, Pascal, ADA…
- Java è un caso particolare, attua infatti una pre-compilazione per ottenere codice intermedio, poi l'interprete esegue tale codice intermedio.

L'esecuzione di un programma compilato è tipicamente più veloce dell'esecuzione di uno interpretato perché la traduzione è effettuata una sola volta prima dell'esecuzione.

Un programma sorgente da interpretare è tipicamente più portabile di un programma da compilare.

Gli **header** file contengono spesso solo delle dichiarazioni, che ci permettono di dichiarare oggetti, occorre però che poi questi oggetti siano definiti da qualche parte, le funzioni e gli oggetti di una libreria sono definiti in ancora altri file (tipicamente pre-compilati) nel caso di linguaggi compilati, in sostanza per fornire una certa libreria vengono forniti sia gli header file, sia i file pre-compilati, contenenti il codice macchina delle funzioni e degli oggetti messi a disposizione dalla libreria.

Quindi, se in un programma includiamo correttamente gli header file di una libreria e ne usiamo le funzioni o gli oggetti, il tutto funziona se: il compilatore **collega** il nostro programma ai file pre-compilati contenenti il codice macchina necessario.

(Per esempio è quello che accade per la libreria di ingresso/uscita; essendo una libreria molto usata, i compilatori sono tipicamente preconfigurati per collegare il programma ai file precompilati di tale libreria).

A volte però il compilatore non può essere predisposto a collegare il nostro programma ai file precompilati di determinare librerie non utilizzate spesso, in quel caso abbiamo bisogno di istruirlo noi passando ad esempio opzioni aggiuntive ( l'abbiamo fatto aggiungendo -lm per la libreria matematica).

Fasi della compilazione:

1) **Preprocessing** : il testo del programma viene modificato in base a delle semplice direttive (es: #include e #define)
2) **Traduzione** (spesso chiamata compilazione, crea confusione tra i termini): genera un programma in linguaggio macchina a partire dal prog. sorgente, il componente di un compilatore è chiamato traduttore, il prog. Generato nella fase di traduzione non è tipicamente eseguibile perché manca il codice delle funzioni e degli oggetti non definiti nel programma stesso, un programma in LM di questo tipo è chiamato **file oggetto.**
3) **Collegamento:** si unisce il file oggetto con i file pre-compilati delle librerie, il risultato è il programma eseguibile; il componente che realizza questa fase si chiama collegatore o linker.

Lo sviluppo di un programma passa per varie fasi: progettazione, scrittura, compilazione, esecuzione, collaudo, debugging…. Si chiama tipicamente **ambiente di programmazione** (o sviluppo) per un dato linguaggio o insieme di linguaggi, l'insieme degli strumenti di supporto alle varie fasi di sviluppo dei programmi scritti con tali linguaggi.

Per scrivere programmi in un linguaggio bisogna conoscere almeno un ambiente di programmazione per tale linguaggio.

**Ambiente di programmazione:**

- **Editor:** serve a creare file di testo, in particolare a creare il programma sorgente.
- **Traduttore (**o compilatore): traduce il programma sorgente scritto in un linguaggio ad alto livello in un programma oggetto.
- **Linker** (collegatore) nel caso il programma richieda l'unione di più modelli, provvede a collegarli per fornire un unico file eseguibile (spesso traduttore e linker, e pre-processore, sono componenti dello stesso compilatore).
- **Interprete** traduce e esegue direttamente ciascuna istruzione del programma sorgente, istruzione per istruzione, ed è alternativo al compilatore
- **Debugger :** consente di eseguire passo-passo un programma controllando via via quel che succede al fine di scoprire ed eliminare errori.

Ambienti di sviluppo: fondamentalmente due tipi:

1) Ambienti dati dalla somma di componenti più o meno indipendenti, un programma per l'editing, uno per la compilazione, uno per il debugging, i sistemi UNIX costruiscono spesso veri e propri ambienti di sviluppo di questo genere (c'è molta scelta per i singoli strumenti di sviluppo).
2) Ambienti di sviluppo integrati (IDE)= ambienti i cui singoli strumenti sono integrati gli uni con gli altri e si dispone di un'unica interfaccia per gestire tutte le fasi (editing, compilazione, esecuzione e debugging).

Il vantaggio principale degli IDE è probabilmente la praticità e semplicità d'uso: interfaccia comune (tipicamente grafica), possibilità di salvare, compilare ed eseguire premendo un solo bottone, posizionamento automatico nei punti del programma in cui si trovano gli errori .

Vantaggi dei sistemi non IDE: si distinguono perfettamente nelle varie fasi dello sviluppo (utilità didattica) e si ha maggiore indipendenza da ogni specifico strumento (si può usare/cambiare lo strumento preferito per ciascuna fase dello sviluppo).

Lezione 20

Classi di memorizzazione

In funzione del suo tempo di vita un oggetto può avere classe di memorizzazione:

- Automatica
- Statica
- Dinamica

**Oggetti automatici:** definiti nei blocchi e parametri formali, creati al raggiungimento della loro definizione durante l'esecuzione del programma, distrutti al termine dell'esecuzione del blocco in cui sono definiti;se non inizializzati hanno valori **casuali** (non dimenticare di inizializzare).

**Oggetti statici:** oggetti globali o definiti nelle funz utilizzando la parola chiave **static** (non vedremo i dettagli di questa parola) creati all'inizio e distrutti al termine dell'esecuzione del programma, se non inizializzati hanno valore **zero.**

**Oggetti dinamici:** allocati nella memoria libera, esistono dal momento dell'allocazione fino alla deallocazione, se non inizializzati hanno valore **casuale**.

Gli oggetti dinamici comportano maggiore difficoltà di gestione rispetto agli oggetti statici ed automatici, maggiore utilizzo della memoria a causa di strutture dati nascoste necessarie per gestirne correttamente l'allocazione e la deallocazione, quindi gli oggetti dinamici vanno utilizzati **solo se** sono chiaramente la soluzione migliore o addirittura l'unica.

Così come gli oggetti, anche le istruzioni stanno in memoria, sono eseguite nell'ordine in cui compaiono, con l'eccezione delle istruzioni di salto che possono far continuare l'esecuzione da una istruzione diversa da quella che la segue. Indirizzo di una funzione: l'indirizzo in cui è memorizzata la prima istruzione della funzione.

Quando una funzione parte, l'esecuzione deve saltare all'indirizzo della funzione, devono essere creati tutti gli oggetti locali della funzione; viceversa quando termina, il controllo torna al **chiamante** ossia alla funzione che conteneva la chiamata alla funzione, il chiamante deve riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione e trovare tutti i suoi oggetti inalterati; tutto ciò accade correttamente grazie ai record di attivazione.

Al momento della traduzione, il compilatore aggiunge del codice che prima dell'esecuzione di una funzione, crea in memoria il corrispondente **record di attivazione** =struttura dati contente gli oggetti locali alla funzione (memorizzati in una zona contigua di memoria).

Per accedere ad un oggetto locale della funzione in esecuzione occorre sapere l'indirizzo a cui inizia il record di attivazione della funzione (=indirizzo base) e distanza dell'oggetto da tale indirizzo (=offset), un registro del processore, chiamato (Stack) Base Pointer (BP) è utilizzato tipicamente per memorizzare l'indirizzo base del record di attivazione della funzione correntemente in esecuzione.

Il compilatore traduce l'accesso ad un oggetto locale con un accesso alla variabile presente all'indirizzo contenuto nel base pointer più l'offset , ogni oggetto locale è quindi di fatto individuato dal suo offset nel record di attivazione.

**Prologo=** codice aggiuntivo inserito dal compilatore per realizzare la chiamata di una funzione, le sue azioni principali sono: creare il record di attivazione, inizializzare il base pointer con l'indirizzo del record di attivazione, una volta effettuata tale inizializzazione, il codice macchina potrà correttamente accedere a tali oggetti.

La dimensione di un record di attivazione varia da una funzione all'altra, ma per una data funzione è tipicamente fissa e calcolabile a priori.

I record di attivazione sono a loro volta memorizzati in una zona della memoria chiamata **stack (pila),** il record di attivazione di una funzione viene **creato dinamicamente** nel momento in cui la funzione viene chiamata, rimane sullo *stack* per tutto il tempo in cui la funzione è in esecuzione (quindi è deallocato solo al termine della funzione.

Funzioni che chiamano altre funzioni danno luogo ad una sequenza di record di attivazione, allocati secondo l'ordine delle chiamate, de-allocati in ordine inverso (i record sono *innestati*).

Nel linguaggio C/C++ tutto si basa su funzioni (anche il main è una funzione), per catturare la semantica delle chiamate annidate è necessario gestire l'area di memoria che contiene i record di attivazione relative alle varie chiamate di funzione proprio come una pila (stack): LIFO last in first out (l'ultimo record ad entrare, è il primo ad uscire).

Nel record di attivazione è sempre presente un collegamento dinamico che contiene l'indirizzo del record di attivazione del chiamante, il quale al ritorno da una funzione permette di sapere l'indirizzo del record di attivazione della funzione che torna in esecuzione; il base pointer può così essere aggiornato con tale indirizzo e accedere con l'offset agli oggetti locali della funzione che torna in esecuzione→ in sostanza si ripristina l'ambiente del chiamante quando la funzione chiamata termina, la sequenza dei collegamenti dinamici costituisce la **catena dinamica** che rappresenta la storia delle attivazioni ("chi ha chiamato chi").

Prima di chiamare una funzione si aggiunge un elemento nel record di attivazione per contenere il *valore di ritorno della funzione*, viene copiato dal chiamante all'interno di tale elemento prima di terminare.

Abbiamo già visto il **prologo (**record di attivazione di una funzione); **epilogo** = è un codice che immette il valore di ritorno in una funzione nel record di attivazione del chiamante e poi dealloca il record di attivazione.

Sia Epilogo che prologo sono inseriti automaticamente dal compilatore nel file oggetto.

Sappiamo che un programmatore genera codice assumendo che venga eseguito da una macchina virtuale che fa vedere la memoria come un contenitore di celle in cui si creano automaticamente variabili con classe di memorizzazione statica e automatica (e di distruggeranno anche automaticamente) e si possono allocare dinamicamente oggetti; per fare ciò il compilatore inserisce del codice aggiuntivo (prologo ed epilogo, chiamate a funzioni del sistema operativo per gestire la memoria dinamica).

**Spazio di indirizzamento** di un processo (processo= programma in esecuzione) = è l'insieme di locazioni di memoria accessibili dal processo (Per schema vedi slide 35 lez_20).

Oggetti statici non inizializzati sono automaticamente posti a zero per questioni di sicurezza e per rendere più deterministico e ripetibile il comportamento dei programmi.

Memoria dinamica e stack possono crescere finché lo spazio libero non si esaurisce, quando un record di attivazione è rimosso o oggetto din deallocato, le locazioni di memoria precedentemente utilizzate **non sono reinizializzate** a 0; ora sappiamo che i valori dinamici ed automatici hanno valori casuali se non inizializzati perché dipende dai valori precedentemente memorizzati in quella zona di memoria.

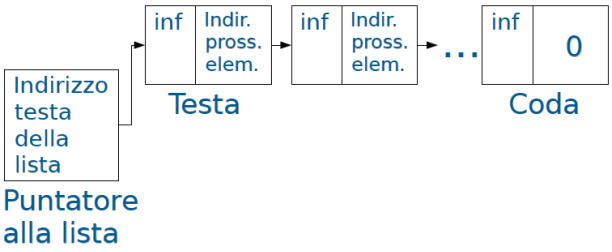
Lezione 21 Liste

In un array invece che allocare e deallocare tutto l'oggetto, potremmo puntare ad uno in particolare, ed insieme ad esso puntare all'indirizzo del prossimo oggetto, e l'ultimo per capire che è l'ultimo, assegniamo valoro dell'indirizzo =0.

**Lista concatenata=** struttura dati i cui oggetti/elementi sono disposti in ordine lineare, diversamente dall'array in cui l'ordine è determinato dagli indici, l'ordine in una lista è determinato da un puntatore in ogni oggetto.

Ciascun elemento contiene un **campo informazioni** ed un campo puntatore (oppure due), il primo elemento di una lista è chiamato **testa (head)** della lista, L'ultimo **coda (tail)** della lista.

Lista singolarmente concatenata o semplice =ciascun elemento contiene solo un puntatore al prossimo elemento.




Il puntatore al prossimo elemento della coda della lista contiene il valore 0 (NULL), il puntatore della testa della lista individua la lista stessa (chiamato anche puntatore alla lista)

Lista doppiamente concatenata o doppia= ciascun elemento contiene sia un puntatore al prossimo elemento che un puntatore all'elemento precedente.

Esistono varie librerie che forniscono il tipo di dato lista, vengono fornite le operazioni di :

- Creazione ed eliminazione
- Inserimento /estrazione di elementi in testa, in fondo, in una data posizione (tipicamente al costo O(1)
- Restituzione del numero di elementi (può costare O(1) oppure O(N))
- Inserimento in ordine (tipicamente al costo O(N))
- Riordinamento (tipicamente al costo O(N logN)

Le funz di libreria si occupano dei puntatori, il programmatore solo del campo informazione; in C++ c'è il tipo di dato *list*, presentato in <list>.

Confronto array-liste:

data una sequenza di N oggetti (ad esempio di tipo **int)**:

Se la sequenza è memorizzata in un array occupa meno spazio in una lista, si può aggiungere un elemento in fondo alla sequenza a costo computazionale inferiore rispetto ad una lista ed infine l'inserimento di un elemento intesta o nel mezzo ha costo O(N).

Se la sequenza è memorizzata in una lista, occupa più spazio rispetto ad un array, si più aggiungere un elemento in fondo a costo computazionale maggiore rispetto ad un array (anche se si dispone di un puntatore all'ultimo elemento), l'inserimento di un elemento in testa alla sequenza ha costo O(1), l'inserimento nel mezzo ha costo O(1) se si conosce l'indirizzo dell'elemento dopo il quale si vuole inserire il nuovo elemento.