

Corso di Laboratorio di Programmazione (Gr.2)

Laura Bozzelli
a.a. 2020/2021

Obiettivi del corso

- Imparare a risolvere problemi computabili in modo algoritmico.
- Imparare a tradurre un algoritmo in programma.
- Imparare a scrivere programmi in linguaggio C.

Cosa il corso NON è!

- Una bacchetta magica con cui diventare subito esperti programmatori
- Un corso dove conoscerete tutti i paradigmi di programmazione strutturata.
- Un corso dove imparerete tutti i segreti del linguaggio C.

Dettagli sul corso

- Corso semestrale con una frequenza di 4 ore la settimana.
- Orari: Lunedì e Mercoledì dalle 14:00 alle 16:00.
- Non si prendono presenze, ma si consiglia di frequentare.
- Luogo: al momento per via telematica tramite la piattaforma Microsoft Teams.

Modalità d'esame

- Compito da fare in laboratorio al calcolatore.
- Prova orale.

Libri di testo

- Il linguaggio C. Principi di programmazione e manuale di riferimento, Brian W. Kernighan, Dennis M. Ritchie.
 - Rappresenta il manuale di riferimento sulla sintassi e semantica del linguaggio C.
- Il linguaggio C. Fondamenti e tecniche di programmazione, Deitel & Deitel.
 - E' anche orientato alle strutture dati e alla composizione dei programmi.

Informazioni sul docente

Laura Bozzelli, Ricercatore INF/01

Dipartimento di Ingegneria Elettrica e delle Tecnologie
dell'Informazione, via Claudio 21.

Email: laura.bozzelli@unina.it

lr.bozzelli@gmail.com

Materiale didattico su web docenti www.docenti.unina.it

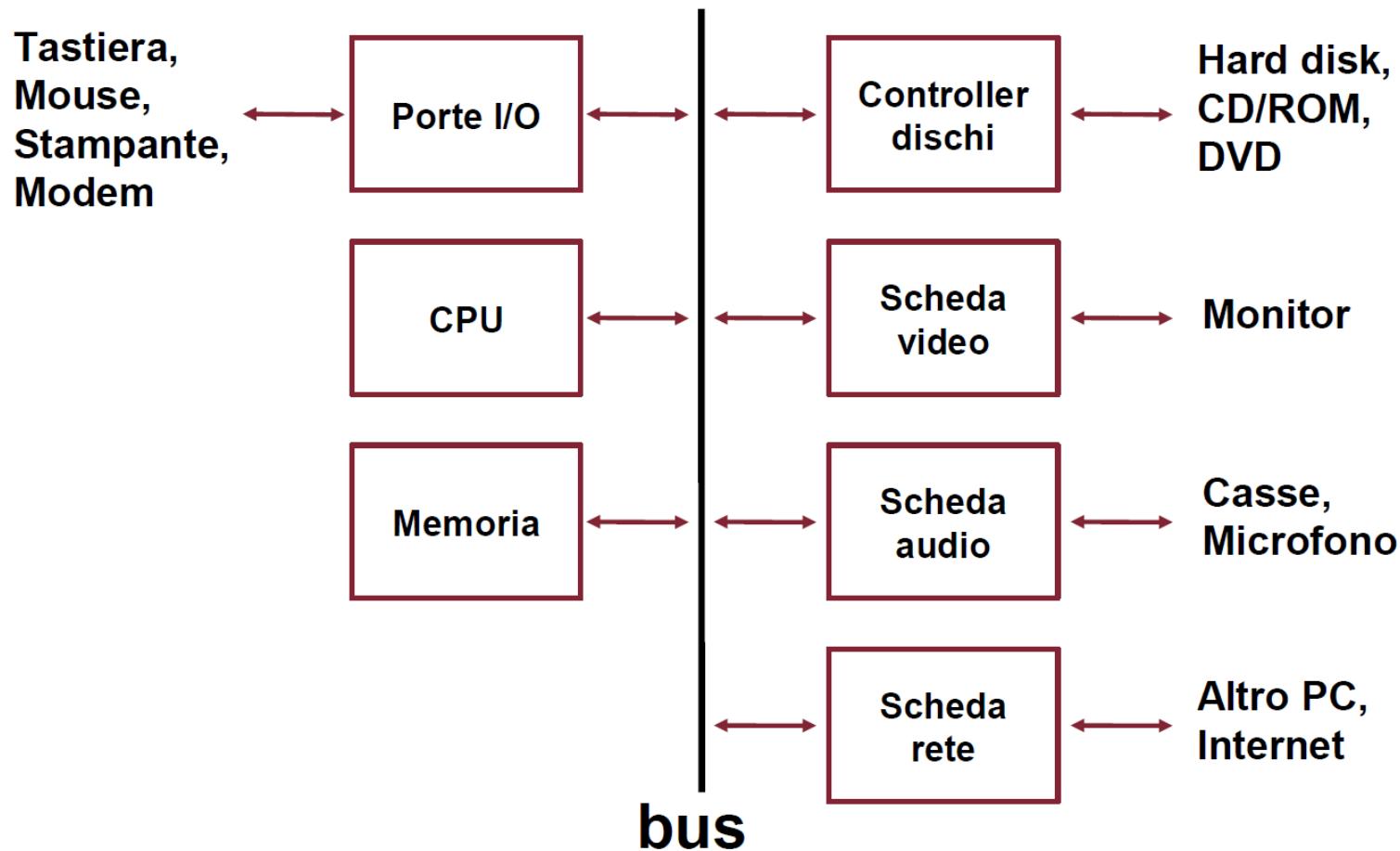
Sommario - Lezione 1: Introduzione

- Architettura dei calcolatori.
- Linguaggi di programmazione.
- Programmi = dati + operazioni.
- Introduzione al linguaggio C.
- Ambiente di sviluppo integrato di programmi in C.
- Scrittura, compilazione, ed esecuzione di un programma in C.

Funzionamento di un computer

- Esegue istruzioni elementari.
- Le istruzioni sono eseguite molto velocemente: un personal computer è in grado di eseguire miliardi di istruzioni in un secondo!
- Elabora dati sotto il controllo di sequenze ordinate di istruzioni chiamate **programmi per computer**.
- Un computer deve essere programmato: i programmi (**software**) caratterizzano il compito che esso svolge.

Schema hardware di un computer



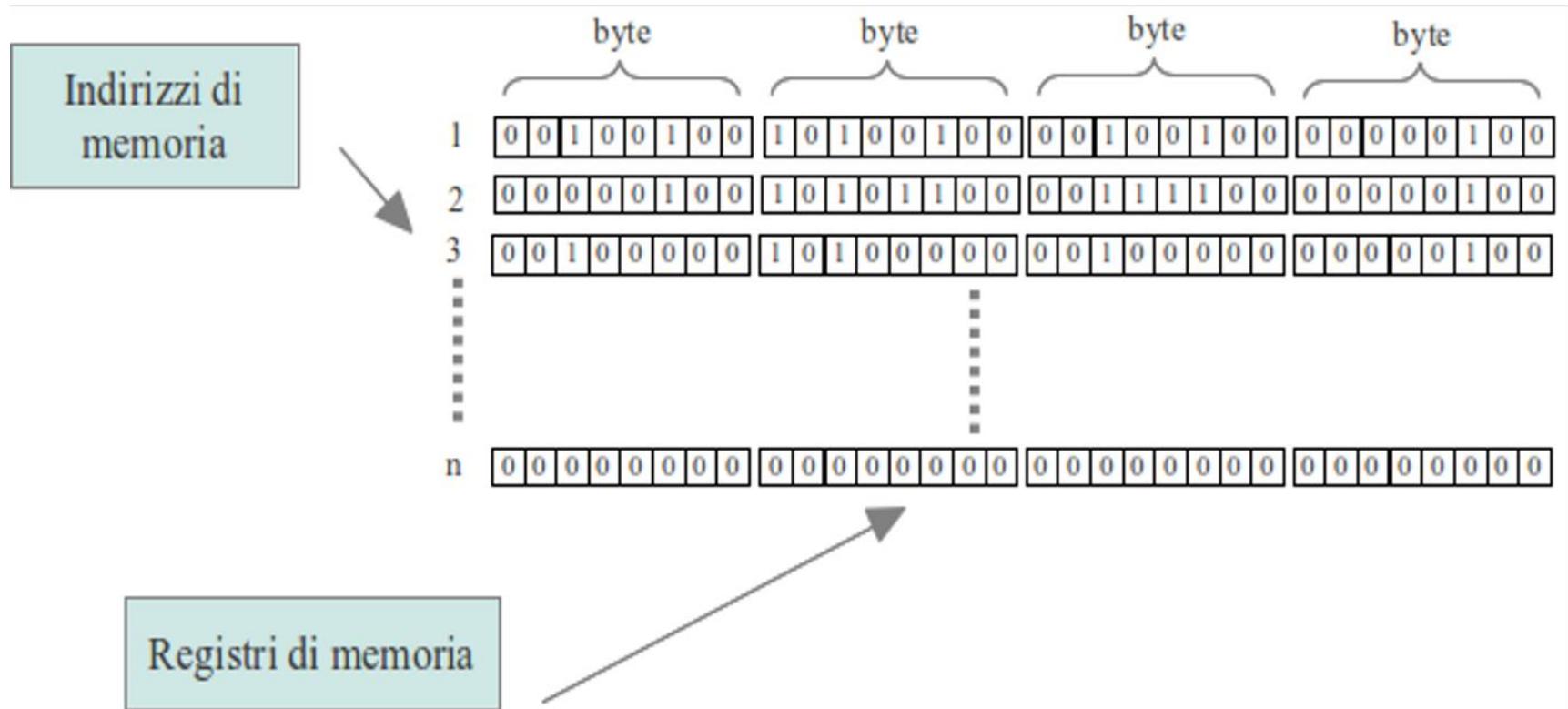
Bus

- Il bus è lo strumento centrale di comunicazione del computer.
- Le unità che vi sono collegate scambiano “informazioni” attraverso il bus.
- Il clock di un bus esprime la velocità con cui vengono scambiati i dati.

Memoria Principale (RAM)

- Memorizza tramite sequenze di byte (1 byte = 8 bit) sia i codici dei programmi in esecuzione sia i valori dei dati che questi programmi usano.
- È una memoria *volatile*: allo spegnimento del computer le informazioni contenute in essa vanno perse.
- Può essere vista (vedi slide seguente) come un elenco ordinato di *registri* (o *celle*) ciascuno avente un indirizzo univoco (un numero intero) che ne consente l'individuazione. Ogni cella consiste di un numero fissato di byte, ad esempio quattro.
- È detta RAM (Random Access Memory): il tempo di accesso ad una cella è indipendente dalla sua posizione (indirizzo). Consente un rapido accesso ai suoi dati.
- Dimensioni tipiche di una RAM di un PC: 2-4 GB (gigabyte = 1 miliardo di byte).

RAM



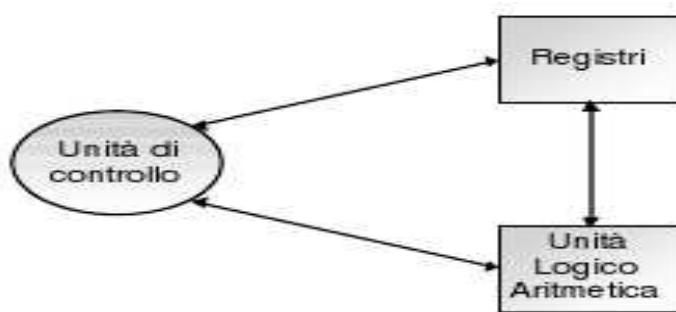
Memoria Secondaria

- La memoria secondaria è un dispositivo, o un insieme di dispositivi, capace di contenere a parità di costo molte più informazioni della memoria principale, a discapito della velocità di accesso ai dati.
- È una memoria *persistente*: le informazioni contenute sono conservate in maniera permanente anche quando il computer viene spento. I programmi o i dati non utilizzati attivamente sono normalmente posti su dispositivi di memoria secondaria.
- Esempi di unità a memoria persistente: dischi rigidi (hard disk: es. 260 GB), chiavette USB (es. 16 GB), e i DVD.

CPU

- L'unità centrale di elaborazione (Central Processing Unit -CPU) è un insieme di circuiti, detto **microprocessore**, che coordina e supervisiona le operazioni delle altre unità del computer.
- Una delle sue funzioni principali è quella di accedere alle istruzioni nella memoria principale, decodificarle ed eseguirle.
- Molti computer moderni hanno CPU multiple e, di conseguenza sono in grado di eseguire molte operazioni simultaneamente.
- Il clock di una CPU, misurato in GHz (ad esempio 1.83 GHz), è la frequenza con cui vengono eseguite le istruzioni elementari (numero di istruzioni elementari eseguite in un secondo).

Componenti della CPU



- Registri con velocità di accesso superiore a quella della memoria principale. Utilizzati per memorizzare risultati parziali durante l'esecuzione di un'istruzione elementare: o l'indirizzo di memoria o il valore di una cella della RAM, oppure il risultato di un'un'operazione applicata ai contenuti di due registri interni.
- Una unità logico-aritmetica (ALU) che esegue operazioni aritmetiche, logiche e di confronto sui dati memorizzati nella RAM e nei registri interni.
- Una unità di controllo che esegue le istruzioni secondo quello che viene detto ciclo fetch-decode-execute (accesso-decodifica-esecuzione).

Ciclo macchina

L'unità di controllo esegue continuamente il ciclo fetch-decode-execute (ciclo macchina), il quale può essere così sintetizzato:

- **Fetch (lettura)**: Acquisizione dalla memoria di un'istruzione del programma.
- **Decode (decodifica)**: Identificazione del tipo di istruzione tra l'insieme delle istruzioni.
- **Execute (esecuzione)**: Esecuzione dell'istruzione.

Esecuzione di un'istruzione macchina

Costituita dai seguenti passi:

- L'indirizzo, chiamiamolo **id_ris**, di un registro della memoria primaria adibito a **risultato** viene acceduto e memorizzato in un registro interno **R**.
- Due registri della memoria primaria adibiti a operandi vengono acceduti e i loro valori, chiamiamoli **val_op1** e **val_op2**, vengono memorizzati in due registri interni **R_1** e **R_2**.
- L'operazione associata all'istruzione viene eseguita sui valori dei due registri **R_1** e **R_2** ed il risultato viene trasferito al registro della RAM con indirizzo **id_ris**, temporaneamente memorizzato nel registro interno **R**.

Unità di input/output

- **Unità di input:** ottengono dati da dispositivi di input e li mettono a disposizione di altre unità di elaborazione quando richiesto dalla CPU.
 - Porte I/O: per ricevere input da tastiere, mouse, dispositivi a memoria secondaria, webcam, ecc.
 - Scheda audio: per ricevere segnali audio da microfoni, strumenti musicali.
 - Scheda di rete: per ricevere dati da Internet o da altri computer.
- **Unità di output:** ottengono dati elaborati dal computer e li inviano a dispositivi di output per poterli utilizzare all'esterno del computer.
 - Porte I/O: per inviare dati a stampanti, dispositivi a memoria secondaria.
 - Scheda audio: per inviare segnali audio ad una periferica d'audio (altoparlanti, cuffie).
 - Scheda video: per inviare dati in uscita da visualizzare su schermi, dispositivi di realtà virtuale (ad esempio, visori HMD come HTC Vive).
 - Scheda di rete: per inviare dati su Internet o ad altri computer.

Linguaggi di programmazione (1)

Le istruzioni di un programma (codice macchina) eseguibile da un calcolatore sono rappresentate come sequenze di byte e coinvolgono semplici operazioni di manipolazione di bit (lettura ed impostazione del valore 0 o 1 di un bit).

- I linguaggi macchina sono estremamente scomodi per i programmatore.
- Tali linguaggi dipendono dall'architettura hardware.
- Per ovviare ai precedenti inconvenienti, vengono utilizzati linguaggi di programmazione più astratti per facilitare la scrittura testuale (specifica) di un programma.

Linguaggi di programmazione (2)

- **Linguaggi assembly**: basso livello di astrazione. Utilizzano sequenze di caratteri alfanumerici per denotare operazioni elementari. Un programma traduttore (assemblatore) è in grado di convertire velocemente programmi assembly in codice macchina.
- **Linguaggi di programmazione di alto livello**: elevato livello di astrazione ed indipendenza dall'hardware.
 - Velocità nel processo di programmazione e facilità nell'implementazione di algoritmi.
 - Un programma traduttore, chiamato **compilatore**, è in grado di convertire un programma di alto livello in linguaggio macchina.
 - Esempi sono il linguaggio C (studiato in questo corso), C++, C# e Java.

Dati e operazioni

Un programma (indipendentemente dal linguaggio di programmazione in cui è scritto) è caratterizzato da due aspetti fondamentali.

- **DATI**

Rappresentazione delle informazioni relative al dominio di interesse.

- **OPERAZIONI**

Manipolazione dei dati che realizzano le funzionalità richieste.

Rappresentazione delle informazioni

- Le informazioni si rappresentano tramite **tipi di dato**.
- Un tipo di dato raccoglie informazioni della stessa tipologia (ad esempio, i numeri interi, i caratteri, le stringhe di caratteri) e, in alcuni casi, anche le operazioni che manipolano tali informazioni (ad esempio, le operazioni aritmetiche sui numeri interi).
- Nei linguaggi di programmazione di alto livello esistono **tipi di dati predefiniti**, ed è possibile definire tipi di dati per rappresentare informazioni specifiche del dominio applicativo.

Classificazione dei tipi di dati

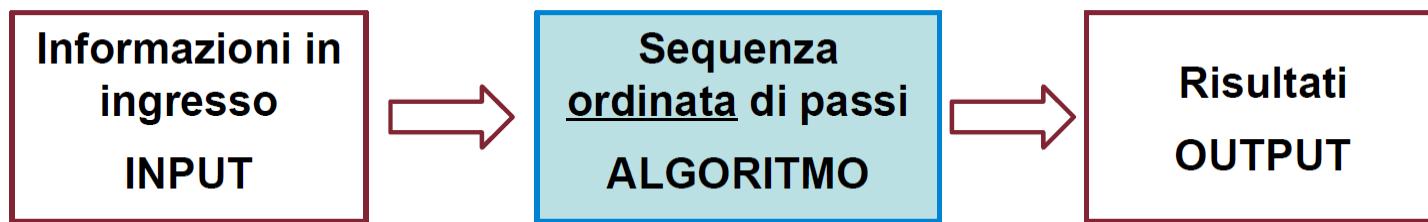
- **Semplici o primitivi:**
 - Es. numeri interi, numeri reali, caratteri.
 - Sono tipi di dati predefiniti.
 - Sono a **lunghezza fissa**: l'insieme delle possibili informazioni rappresentabili è finito. Ad esempio un tipo intero rappresenta solo un sottoinsieme finito di tutti i numeri interi.
- **Complessi:**
 - Sono definiti a partire da dati primitivi.
 - Es. vettori, matrici, liste.
 - Possono essere o predefiniti o definiti dal programmatore.
 - Possono essere a **lunghezza fissa** o a **lunghezza variabile**. In quest'ultimo caso l'insieme delle informazioni rappresentabili è infinito (ad esempio, le stringhe di caratteri).

Implementazione delle operazioni

- In genere si specificano delle operazioni quando si risolve un problema.
- La risoluzione di un problema tramite l'ausilio di un computer si basa sulla definizione di un procedimento di ragionamento che può essere automatizzato (**algoritmo**).

Algoritmo: possibile definizione

Procedimento risolutivo attraverso il quale otteniamo la soluzione ad un problema, ovvero un **insieme di passi** che, **eseguiti in ordine**, permettono di calcolare i risultati a partire dalle informazioni date in ingresso.



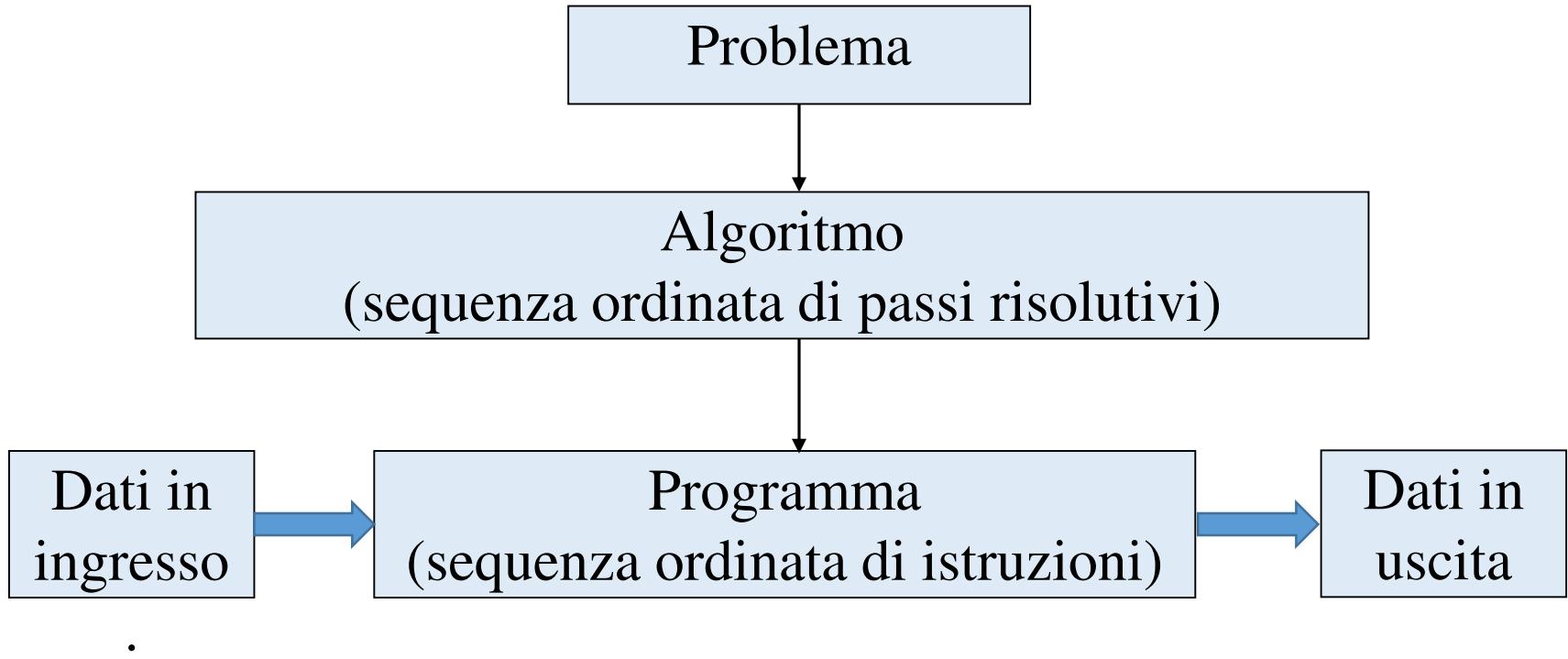
Esempio: Dato un nominativo trovare il numero corrispondente da un elenco telefonico. Un possibile algoritmo è scandire sequenzialmente i nominativi presenti nell'elenco fino a trovare quello cercato. **È un algoritmo efficiente?**

Algoritmo: proprietà.

- **Non ambiguità**: le istruzioni devono essere univocamente interpretabili dall'esecutore
- **Eseguibilità**: ogni istruzione deve poter essere eseguita (in tempo finito) con le risorse a disposizione.
- **Terminazione**: l'esecuzione dell'algoritmo deve terminare in tempo finito per ogni insieme di dati in ingresso.

Codifica

Una volta individuato l'algoritmo, questo va **codificato** nel linguaggio di programmazione prescelto.



Paradigmi di programmazione

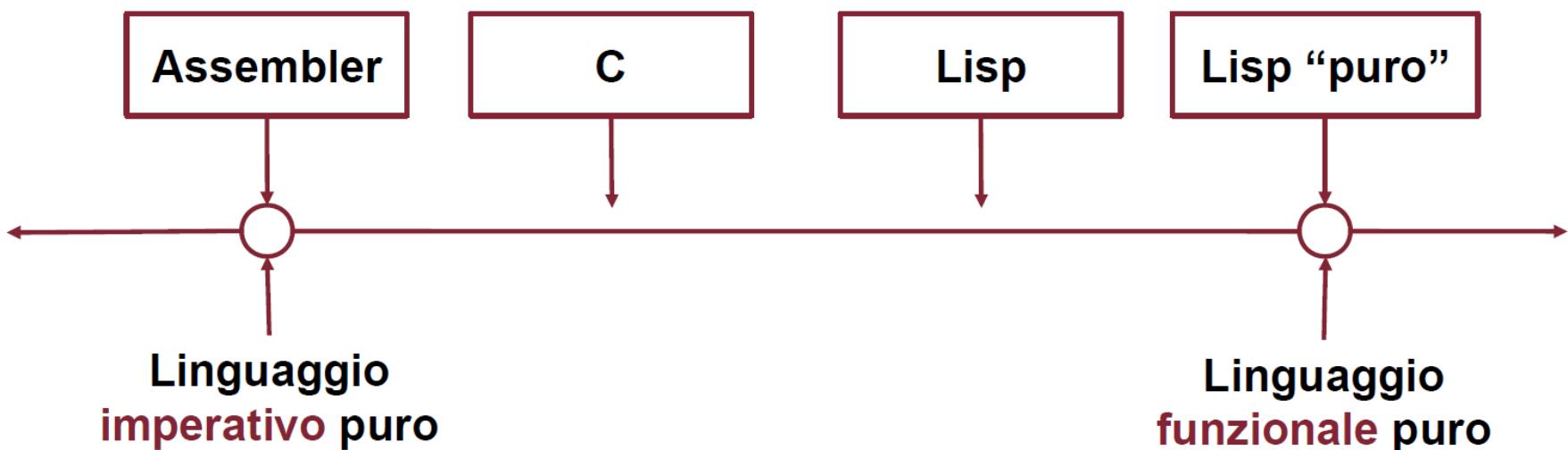
Esistono tre principali paradigmi di programmazione che si distinguono per l'enfasi che pongono sui due aspetti fondamentali: dati (oggetti) e operazioni.

- **IMPERATIVO**: enfasi sulle operazioni intese come azioni, comandi, istruzioni che modificano lo stato (valori) degli oggetti.
- **FUNZIONALE**: enfasi sulle operazioni intese come funzioni che calcolano risultati; gli oggetti sono funzionali alla elaborazione.
- **ORIENTATO AGLI OGGETTI**: enfasi sugli oggetti che complessivamente rappresentano il dominio di interesse; le operazioni sono funzionali alla rappresentazione.

I linguaggi di programmazione forniscono supporto (in misura diversa) per i vari paradigmi.

Linguaggio C (1/2)

Linguaggio di programmazione di **alto livello** che supporta i paradigmi di programmazione **imperativo** e **funzionale**.



Linguaggio C (2/2).

- Il C nasce dall'idea di avere un linguaggio ad alto livello che permettesse di scrivere in modo efficiente un sistema operativo.
- Largamente utilizzato per sviluppare applicazioni che richiedono prestazioni elevate: sistemi operativi, sistemi real time, e sistemi di telecomunicazione.
- Il C ha avuto un lungo processo di standardizzazione che si è concretizzato nello standard ANSI C.

Perchè utilizzare il C ?

- Molto vicino alla struttura di un calcolatore.
- Può maneggiare attività di basso livello (Esplicita gestione della memoria principale: manipolazione di byte ed indirizzi)
- Ha strutture di alto livello che consentono modularità e riutilizzo del codice.
- Punto di partenza per linguaggi object oriented, quali C++ e Java.
- Permette di generare programmi molto efficienti.

Aspetti dei dati in C.

Nel linguaggio C, ad ogni dato (chiamato anche oggetto) sono associati i seguenti aspetti:

- **Nome simbolico**: chiamato anche **identificatore**. Rappresenta una stringa di caratteri utilizzata nel programma per riferirsi alla specifico dato.
- **Indirizzo**: un dato è memorizzato nella memoria principale in una sequenza contigua di registri. Per indirizzo ci si riferisce all'indirizzo di memoria del primo registro di tale sequenza.
- **Tipo di dato**.
- **Valore**: rappresenta il valore del dato ad un certo stato dell'elaborazione, corrispondente al contenuto dei registri di memoria associati al dato (ad esempio, un dato di tipo intero può assumere i valori 1, -1, ecc..). Un dato il cui valore può cambiare nel corso dell'esecuzione di un programma prende il nome di **variabile**. D'altra parte, un dato il cui valore non cambia prende il nome di **costante**.

Dichiarazione di dati in C.

Nel linguaggio C, per specificare che si vuole utilizzare un dato di un certo tipo si utilizza una **dichiarazione** (**dichiarazione di variabile** o **dichiarazione di costante**), e, cioè, un’istruzione in cui si fornisce:

- Nome simbolico (identificativo) del tipo di dato.
- Identificativo del dato.
- Opzionalmente, un valore iniziale del dato.

Ad esempio, l’istruzione

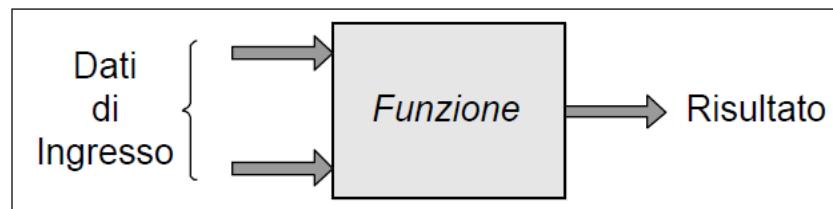
```
int contatore = 3;
```

dichiara una variabile di nome ‘contatore’ di tipo **int** (**int** è il nome di un tipo intero predefinito del linguaggio C) che assume il valore iniziale 3.

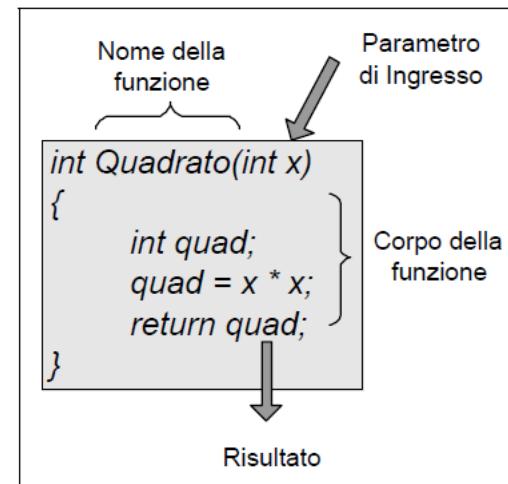
Implicitamente, l’istruzione indica anche che deve essere riservata una area della memoria principale per memorizzare la variabile ‘contatore’.

Struttura di un programma in C (1/2).

Un programma in linguaggio C è costituito da moduli chiamati **funzioni**. Idealmente, una funzione prende dati in ingresso e restituisce un risultato in uscita.



Una funzione in linguaggio C ha un **nome** (identificativo), un **elenco di parametri di ingresso**, un **corpo** (sequenza di istruzioni dove i parametri di ingresso vengono utilizzati per produrre un risultato in uscita), ed un **risultato di uscita**. Un esempio, è la funzione 'Quadrato' alla destra che prende in ingresso un numero intero x e restituisce in uscita x^2 .



Struttura di un programma in C (2/2).

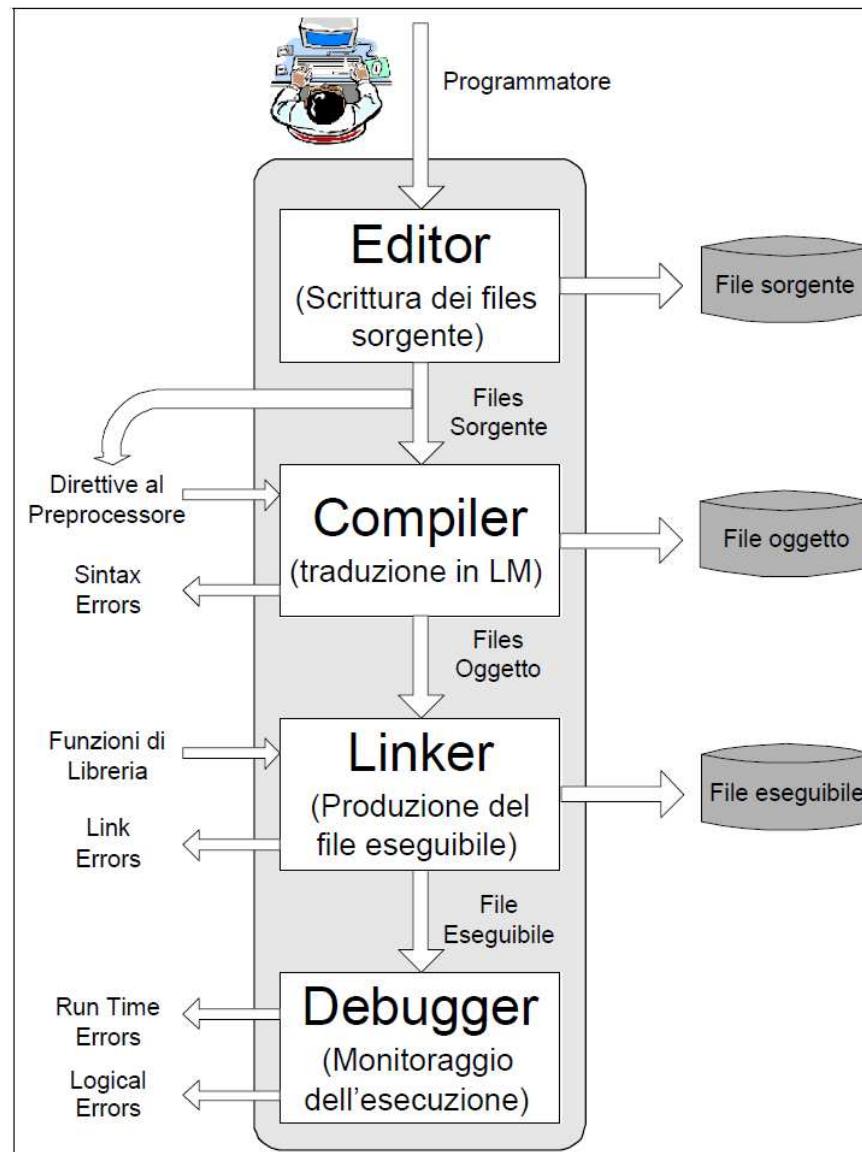
- Ogni programma in C deve contenere una funzione particolare, chiamata **main**, che indica il punto di inizio del programma (la prima istruzione della funzione è la prima eseguita nel programma) ed i cui parametri vengono specificati dall'utente quando fa partire il programma.
- Nel corpo della funzione **main** vengono richiamate altre funzioni, che a loro volta possono usarne altre ancora, e così via.
- Oltre alle funzioni definite direttamente nel programma, è possibile utilizzare funzioni di librerie già compilate, ed, in particolare, le funzioni della libreria standard del C.

Ambiente di sviluppo di programmi in C.

Come supporto alla programmazione in C useremo un **ambiente di sviluppo integrato (IDE)** per il C, un software per scrivere ed eseguire programmi in C che supporta la **libreria standard del C** ed integra le seguenti funzionalità:

- **Editor di testo** per scrivere e salvare su disco i programmi in file di testo (**file sorgente**).
- **Compilatore**: programma per tradurre il codice C di un file sorgente in codice macchina salvato su disco in un **file oggetto**.
- **Linker**: collega i codici oggetto alle librerie, genera un file eseguibile e lo memorizza su un disco.
- **Debugger**: programma per monitorare l'esecuzione di un programma al fine di individuare eventuali errori.

Sviluppo di programmi in C.



Scrittura di programmi in C.

- Per scrivere un programma si utilizza l'editor di testo fornito dall'ambiente di sviluppo integrato (IDE).
- Un editor di testo fornisce funzionalità per indentare automaticamente il codice e per evidenziare con colori diversi parti del codice sintatticamente distinte (costanti, identificatori, ecc..).
- Il programma scritto viene salvato in un file di testo (**file sorgente**) su un dispositivo di memoria secondaria, come un disco fisso. I file sorgente devono terminare con l'estensione **.c**.
- Le funzioni che compongono un programma possono essere scritte in file separati (multipli files sorgenti). L'IDE utilizza un **progetto** per gestire i diversi file sorgente che contengono il programma.

Fase di precompilazione in C.

Quando in un ambiente IDE viene avviata, tramite un opportuno comando, la compilazione di un programma C viene eseguito automaticamente un programma di **prelaborazione** prima che inizi la fase di traduzione da parte del compilatore.

- Il **preprocessore** (o **precompilatore**) esegue speciali comandi presenti nel programma chiamate **direttive al preprocessore** (tali comandi iniziano con il simbolo #).
- Le direttive al preprocessore non rappresentano istruzioni del programma ma indicano che certe operazioni devono essere eseguite sul programma prima della compilazione. Queste operazioni solitamente consistono nell'inclusione di altri file nel file da compilare (direttive **#include**) e nell'esecuzione di varie sostituzioni di testo.
- I file indicati dalle direttive **#include** si chiamano **file header** e hanno l'estensione **.h**. Tali file contengono le dichiarazioni dei **prototipi o interfacce** di funzioni di altri file sorgente o della libreria standard del C (o anche di librerie di terze parti).

Fase di compilazione in C.

- I files sorgente del programma in C vengono compilati uno ad uno per produrre **files oggetto** (file con estensione **.o**). Un file oggetto è la traduzione in linguaggio macchina delle funzioni in C scritte nell'associato file sorgente.
- Le funzioni codificate in un file sorgente possono contenere riferimenti a funzioni definite in altri files sorgente o appartenenti alla libreria standard del C (o anche a librerie di terze parti). Il file oggetto prodotto dal compilatore contiene solitamente ‘buchi’ dovuti a queste parti mancanti. Dunque **un file oggetto non rappresenta un file eseguibile**.
- Nella fase di compilazione si rilevano **errori sintattici (errori di compilazione)** eventualmente presenti nel programma. In presenza di istruzioni che violano le regole sintattiche del linguaggio, la fase di compilazione termina con insuccesso (i file oggetto non vengono creati). Per aiutare ad individuare e a correggere le istruzioni scorrette, il compilatore fornisce opportuni messaggi di errore.

Fase di linking in C.

- Compito del linker è di collegare tra di loro i vari file oggetto per produrre un **file eseguibile** contenente il codice macchina del programma.
- Per ogni parte mancante di un codice oggetto che si riferisce ad una chiamata di funzione definita altrove, il linker localizza il codice oggetto associato alla funzione e sostituisce la parte mancante con istruzioni macchina appropriate.
- In questa fase vengono rilevati degli errori relativi alle chiamate di funzione (**errori semanticci**): il linker si accorge se viene chiamata una funzione che non esiste o se viene invocata una funzione con parametri diversi da quelli che ci si attende.

Debug di programmi.

- Un programma può essere compilato ed eseguito in un **ambiente di debug** per individuare **errori a tempo di esecuzione** (il programma è sintatticamente corretto e le chiamate di funzione sono a posto, ma non fa quello che dovrebbe fare).
- Il debug si basa sull'impiego di tool che consentono di arrestare automaticamente l'esecuzione del programma dopo l'esecuzione di alcune istruzioni. In pratica, si esegue il programma passo per passo, monitorando i valori delle variabili alla ricerca di valori strani o non previsti. L'interruzione dell'esecuzione può avvenire passo passo ad ogni singola istruzione, oppure in punti esplicitamente assegnati dall'utente (**breakpoint** o **watchpoint**).
- In un ambiente di debug, il compilatore introduce ulteriori istruzioni nel codice oggetto per consentire le funzionalità di debug.

Lezione 2

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 2: IDE per C

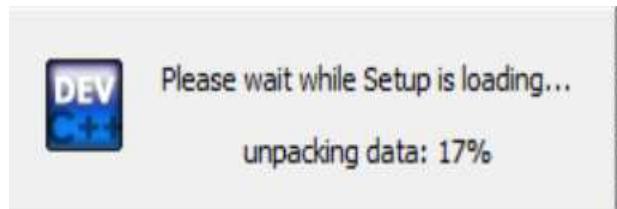
- Scaricare Dev-C++.
- Struttura dell'ambiente Dev-C++.
- Primo programma.
- Compilazione, linking, ed esecuzione.
- Esame del primo programma.
- Compilazione e linking con errori.
- Tipi di dati semplici in C.

Strumenti per lo sviluppo in C

- Sistema operativo Windows 10
- IDE Dev-C++ versione 5.11. Scaricare il file eseguibile di installazione su
<https://sourceforge.net/projects/orwelldevcpp/>

Installazione IDE Dev-C++ (1)

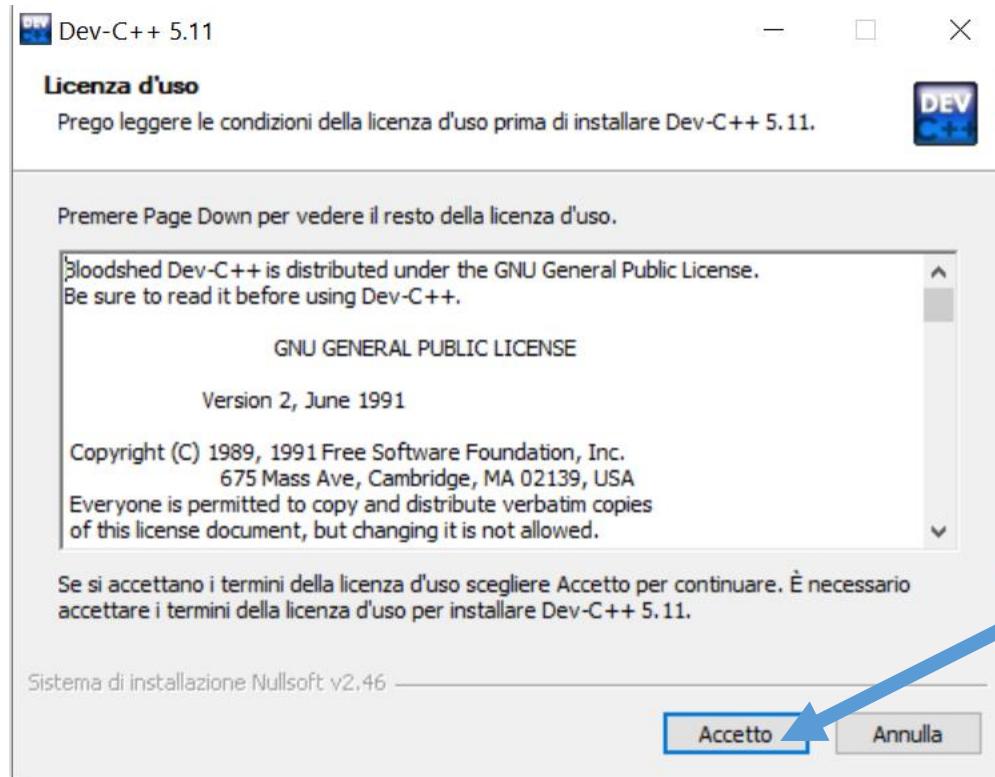
- Il file eseguibile di installazione è **Dev-Cpp 5.11 TDM-GCC 4.9.2 Setup.exe** (versione 5.11 per Windows 10).
- Fare doppio click sul nome del file per eseguirlo.



**Scegliere Italiano
Cliccare su OK**

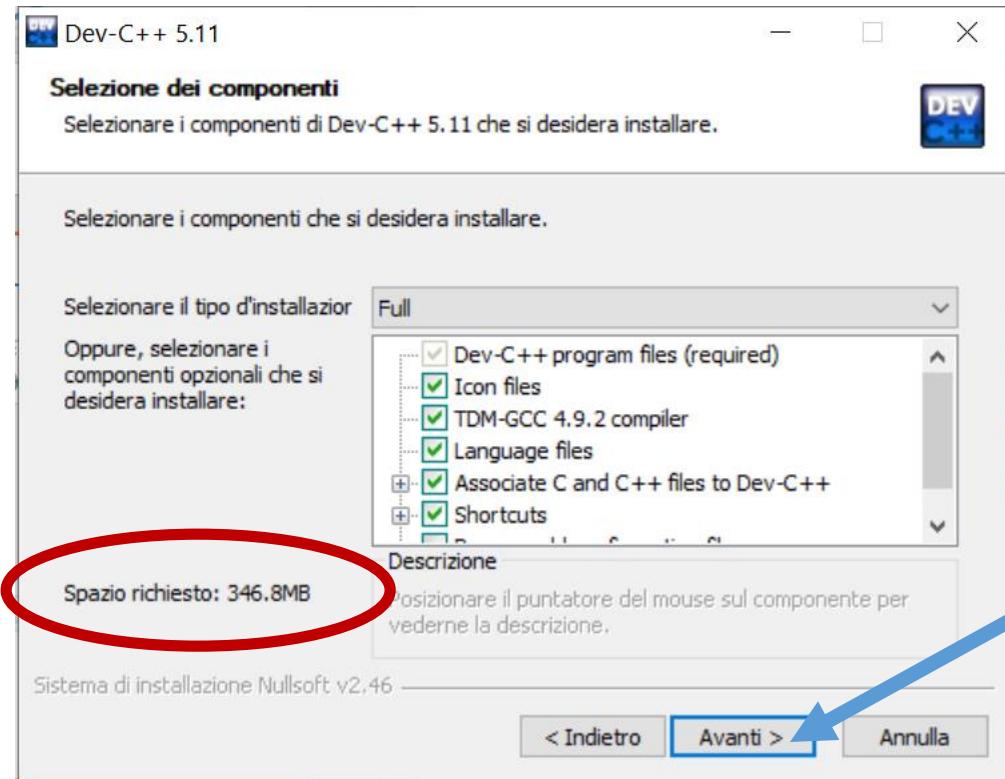


Installazione IDE Dev-C++ (2)



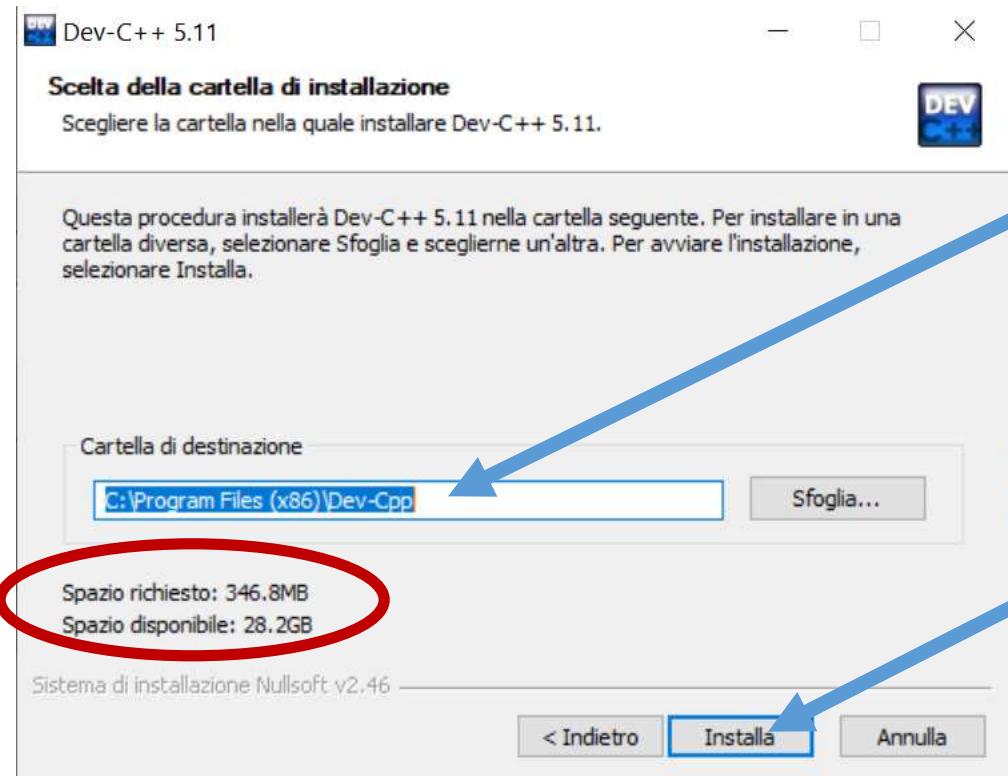
Cliccare su Accetto

Installazione IDE Dev-C++ (3)



Cliccare su Avanti

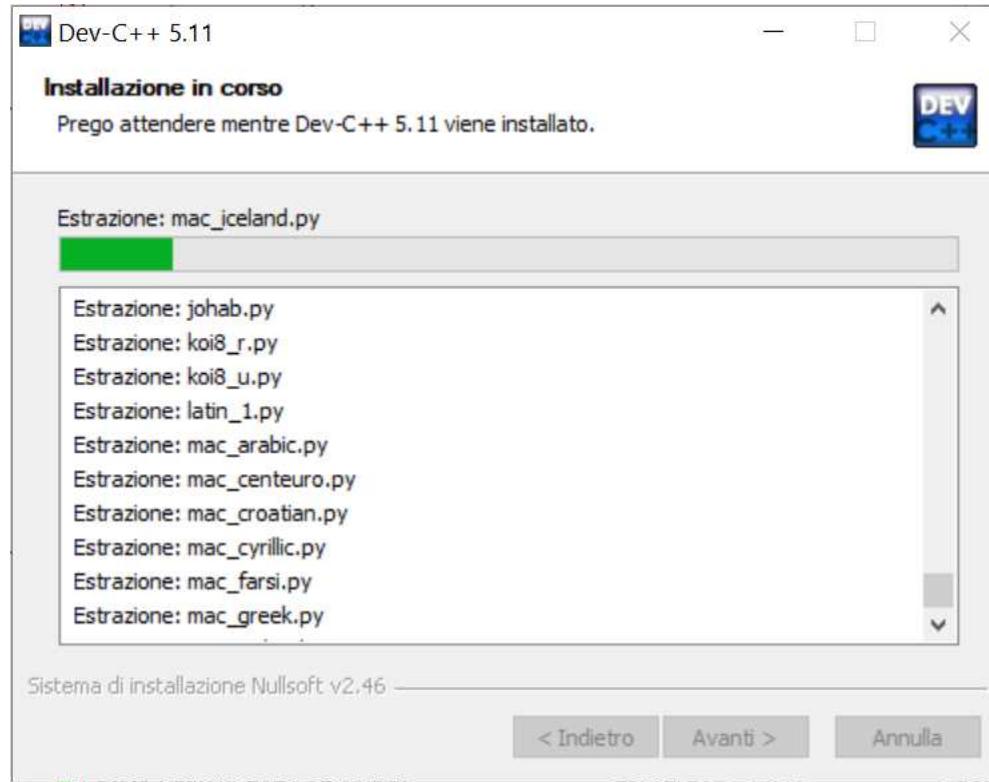
Installazione IDE Dev-C++ (4)



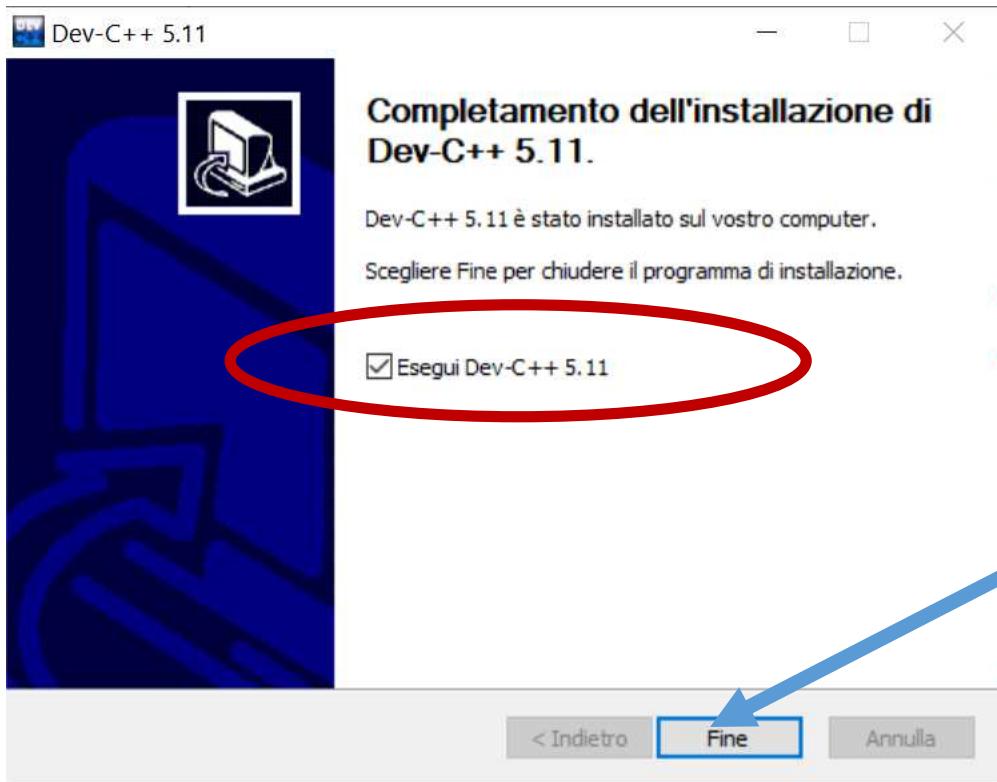
Cartella di installazione

Cliccare su Installa

Installazione IDE Dev-C++ (5)



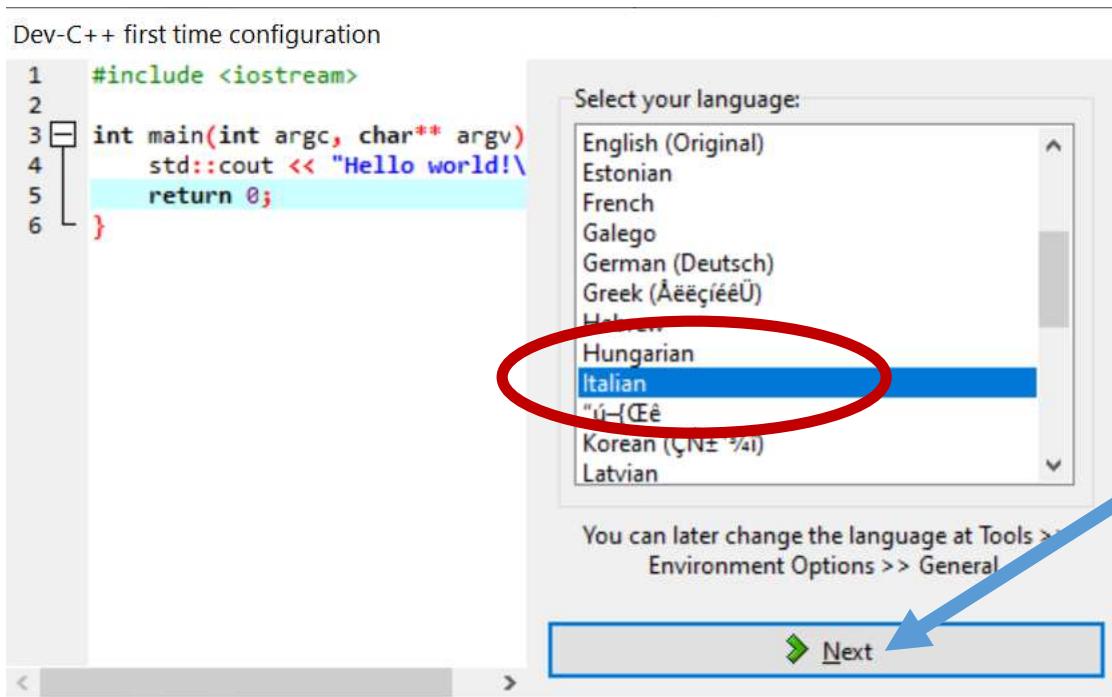
Installazione IDE Dev-C++ (6)



Clicare su Fine

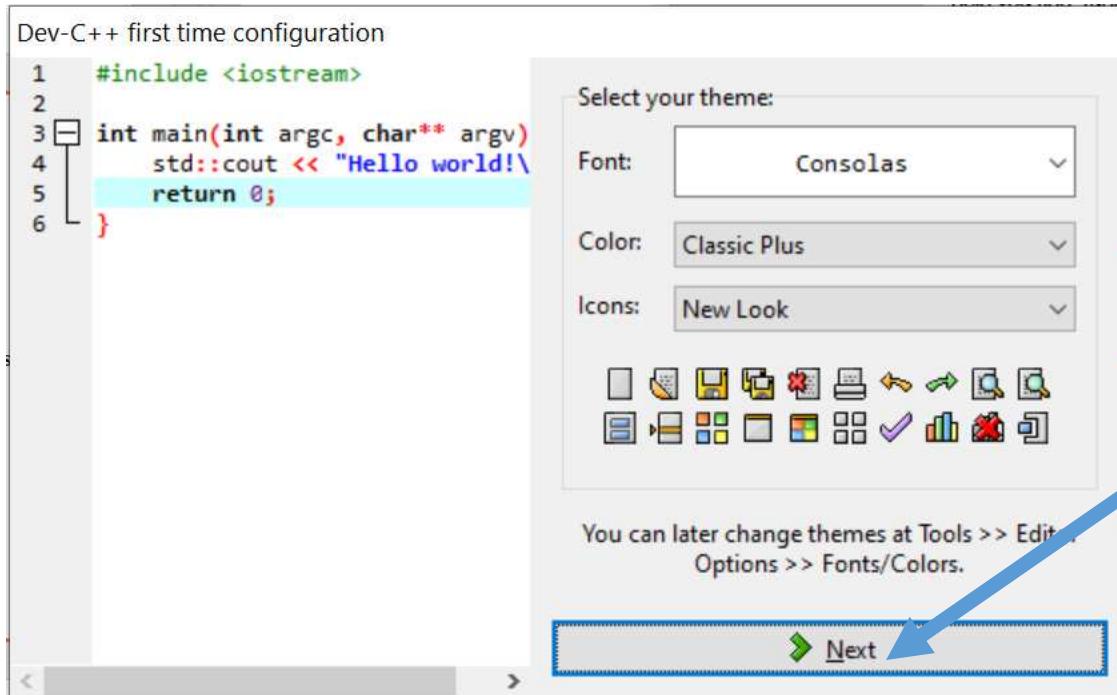
Installazione terminata.

Configurare Dev-C++ al primo avvio (1)



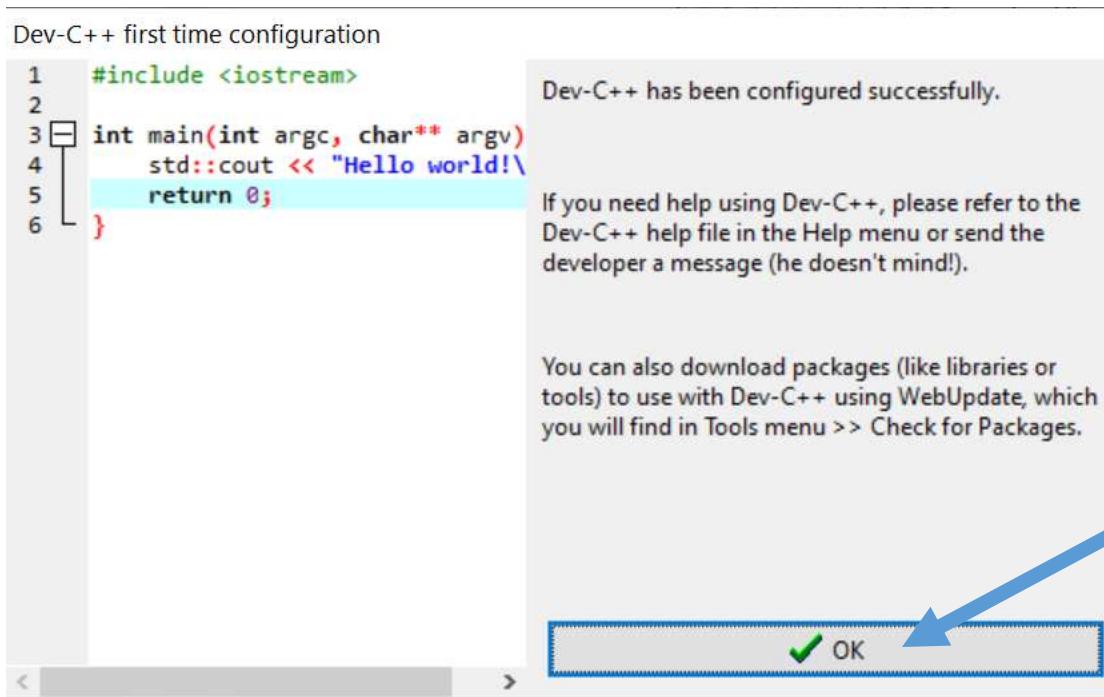
Cliccare su Next

Configurare Dev-C++ al primo avvio (2)



Cliccare su Next

Configurare Dev-C++ al primo avvio (3)

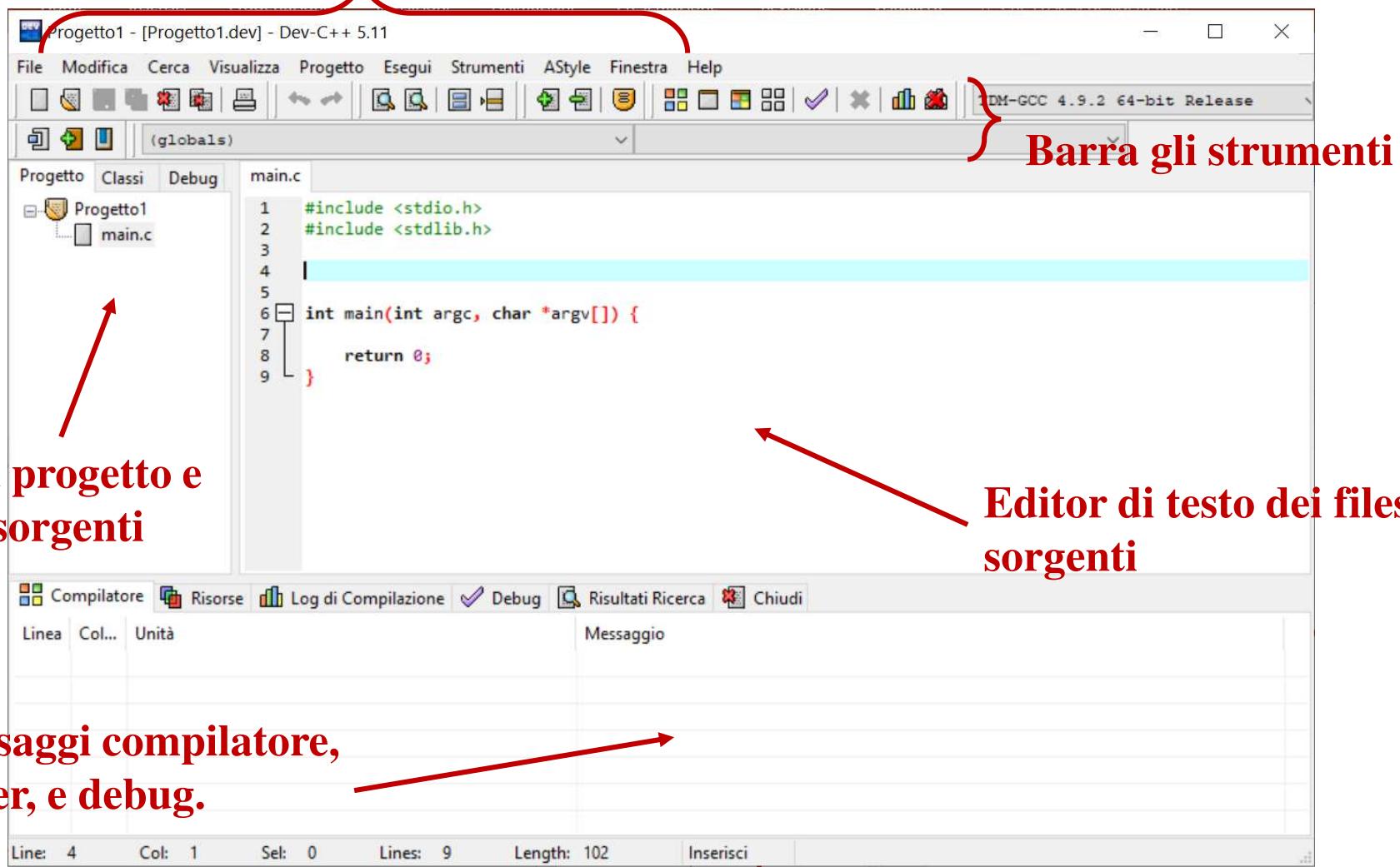


Cliccare su OK

**Ora è tutto pronto
per iniziare!**

Finestra dell'applicazione Dev-C++ (1)

Barra dei menu



Vista progetto e
files sorgenti

Barra gli strumenti

Editor di testo dei files
sorgenti

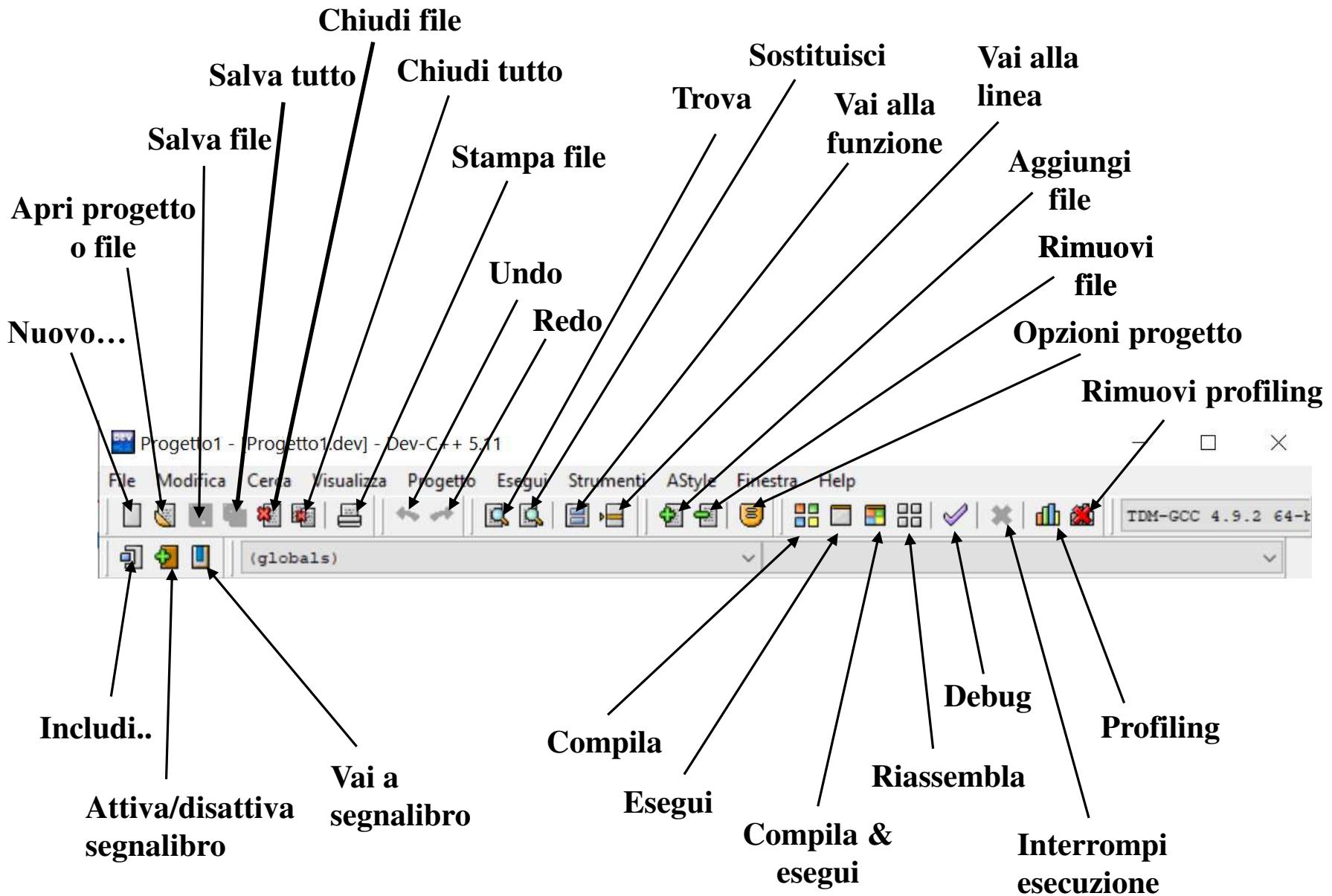
Messaggi compilatore,
linker, e debug.

Finestra dell'applicazione Dev-C++ (2)

Suddivisa in diverse zone:

- Zona nella parte superiore per visualizzare la barra dei menu e la barra degli strumenti i cui pulsanti forniscono un'alternativa per attivare le funzionalità associate alle voci dei menu.
- Zona nella parte centrale a sinistra per visualizzare i file sorgenti del progetto corrente.
- Zona nella parte centrale a destra dove vengono editati i file sorgenti.
- Zona nella parte inferiore per leggere informazioni restituite dal compilatore e dal linker, informazioni di debug, e risultati di ricerca nei file sorgenti.

La barra degli strumenti



Comandi principali (1/2)

- **Nuovo Progetto** e **Apri progetto**: per creare un nuovo progetto o aprirne uno precedentemente creato e salvato su disco. Un file di progetto è un file con estensione **.dev**.
- **Nuovo File**, **Salva File**, e **Chiudi File**: per creare, salvare e chiudere file sorgenti associati al progetto corrente.
 - I file sorgenti **aperti** sono visualizzati nella zona delle finestre di editing (chiamiamo **attivo** il file che si sta editando).
 - Quando si chiude un file, questo continua a far parte del progetto e viene compilato e linkato come i file aperti.
 - Quando si crea un file questo viene aggiunto al progetto ma **non** viene salvato.
- **Stampa File**: per stampare il file attivo.
- **Undo** e **Redo**: il primo comando annulla le operazioni effettuate (dalla più recente all'indietro), il secondo commando ripristina le operazioni annullate (dalla meno recente in avanti).

Comandi principali (2/2)

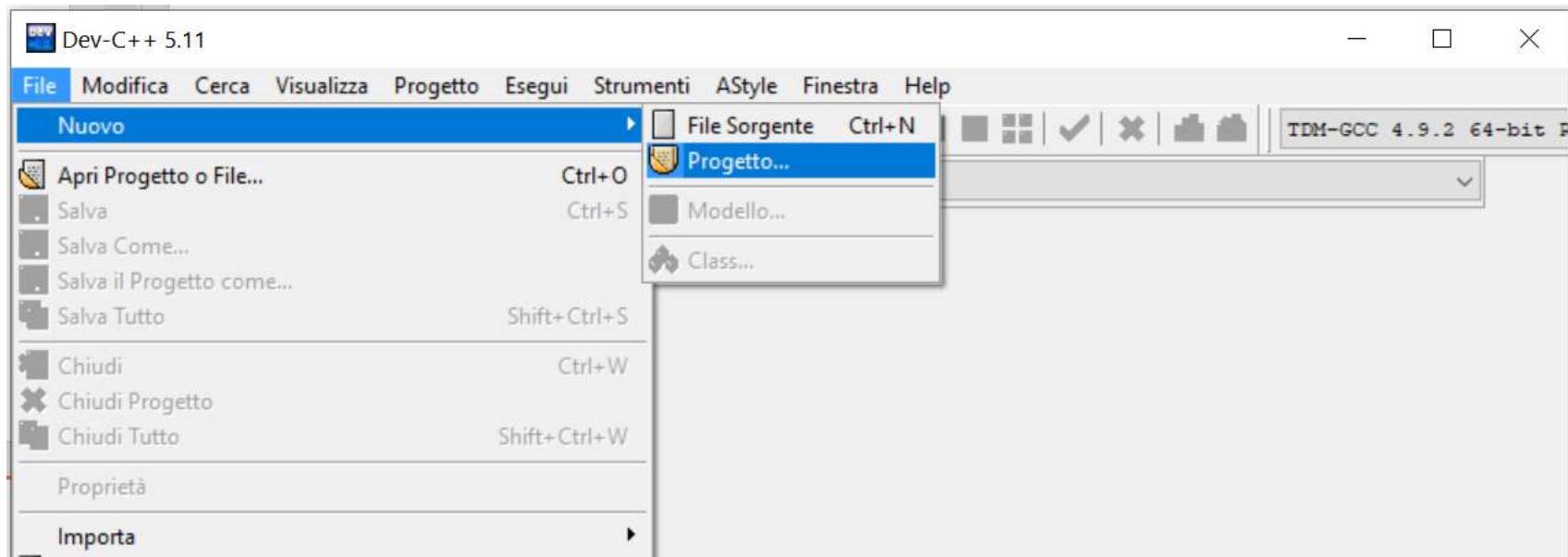
- **Trova** e **Trova successivo**: consentono di cercare delle stringhe nel file sorgente attivo o nei file aperti o nei file di tutto il progetto.
- **Vai alla linea**: consente di posizionare il cursore del file sorgente attivo alla riga indicata in un'apposita finestra di dialogo.
- **Compila, Esegui, Compila & Esegui, Debug**: per eseguire operazioni di compilazione, esecuzione, e debug del codice.
- **Aggiungi File ed Elimina File**: per aggiungere un file sorgente (già creato e salvato su disco) al progetto e per eliminare un file dal progetto. Quando un file viene eliminato esso non verrà più coinvolto nelle operazioni di compilazione e linkaggio del progetto, ma esso **NON** viene fisicamente cancellato dal disco.
- **Inserisci..**: apre un menu dal quale è possibile inserire nella posizione corrente del cursore (nel file sorgente attivo) o un commento di intestazione, o la data e l'ora, o una funzione **main** (viene generata con una sola istruzione: **return 0**).

Realizzazione di un programma

- Nell'ambiente di sviluppo si può realizzare un programma costituito da diversi files sorgente. L'ambiente utilizza un **progetto** per gestire i diversi files del programma.
- In questo corso ci concentreremo su programmi C per creare **applicazioni a console**, ovvero concepite per un'interazione testuale con l'utente (senza quindi l'utilizzo di interfacce grafiche tipo Windows).

Creazione di un nuovo progetto (1/4)

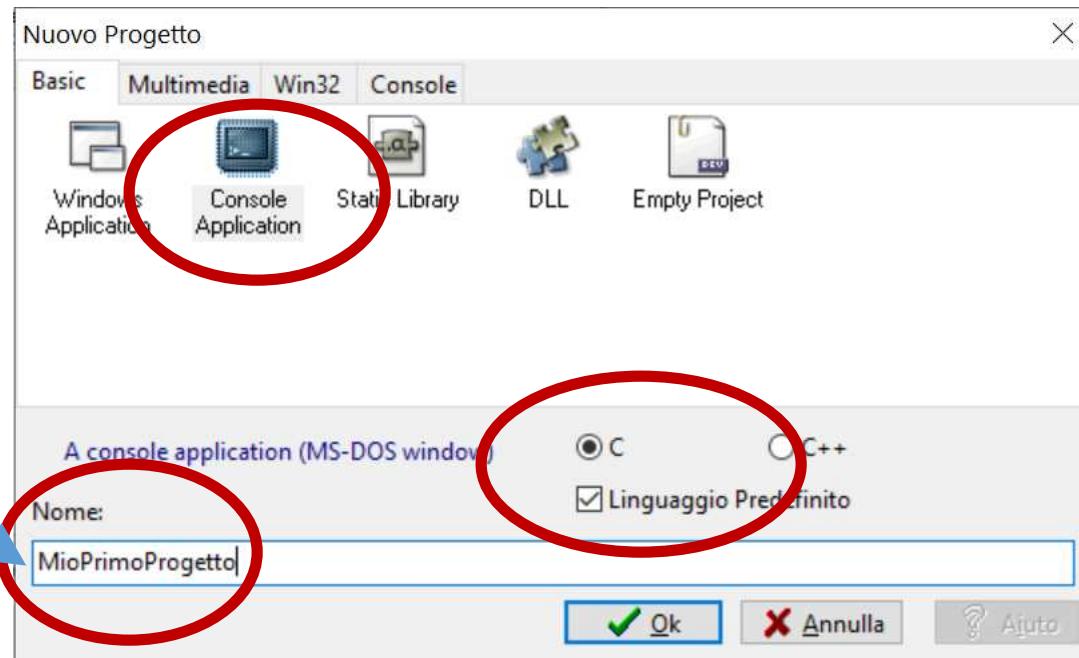
Selezionare il comando **Nuovo Progetto**. Viene visualizzata un finestra di dialogo (vedi slide seguente).



Creazione di un nuovo progetto (2/4)

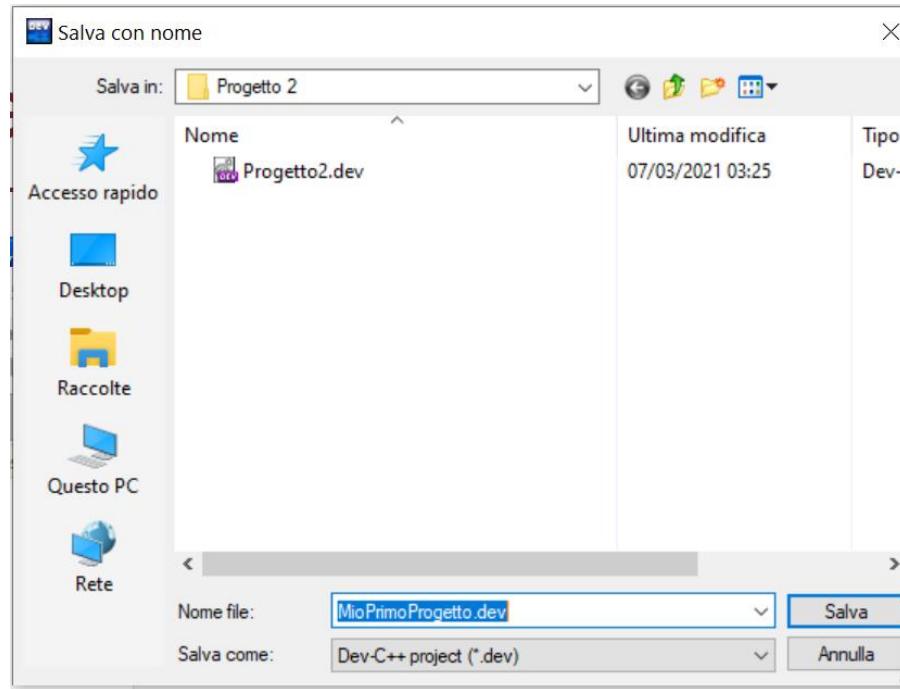
Cliccando sul pulsante **OK** viene visualizzato un **file browser** (vedi slide seguente).

**Scegliere
un nome**



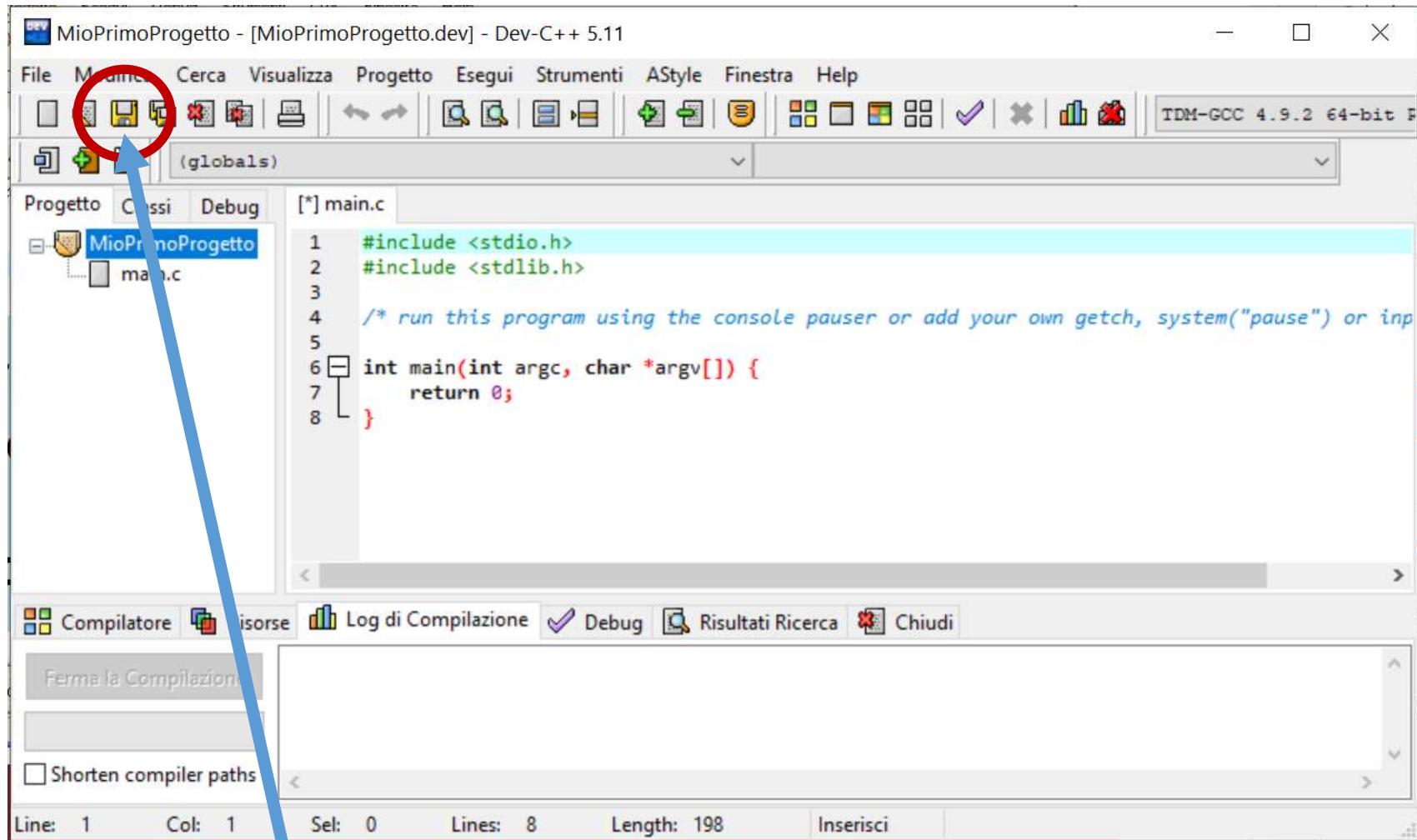
Creazione di un nuovo progetto (3/4)

Scegliere un cartella sul disco dove posizionare il file di progetto .dev e cliccare sul pulsante OK per terminare la procedura di creazione di un nuovo progetto.



La procedura crea un nuovo progetto con un solo file dal nome ‘**main.c**’. Questo file sorgente contiene solamente il *template* di base per la creazione di un semplicissimo programma (vedi figura seguente).

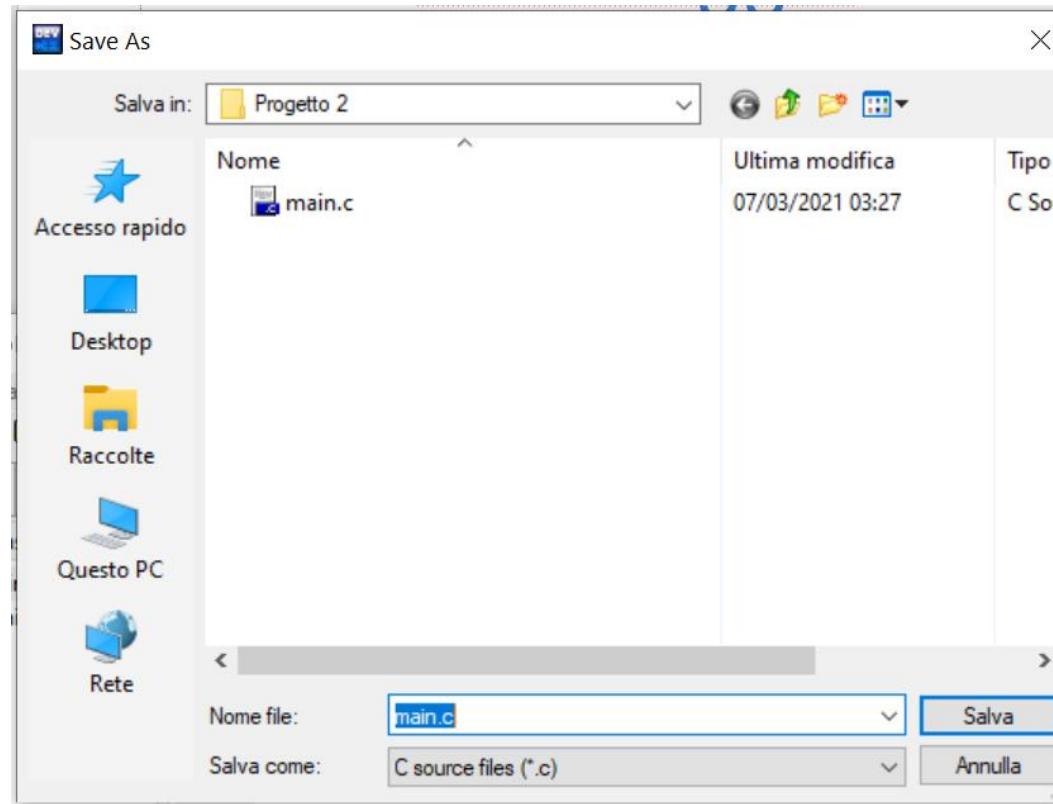
Creazione di un nuovo progetto (4/4)



Il file sorgente 'main.c' deve essere salvato su disco. Per fare ciò cliccare sul pulsante '**Salva File**' della barra degli strumenti.

Salvataggio del file sorgente

Eventualmente rinominare il file sorgente, e scegliere un cartella sul disco dove posizionare il file sorgente (è consigliabile porlo nella stessa cartella contenente il file di progetto o in una sua sottocartella). Cliccare sul pulsante OK per terminare la procedura di salvataggio del file.

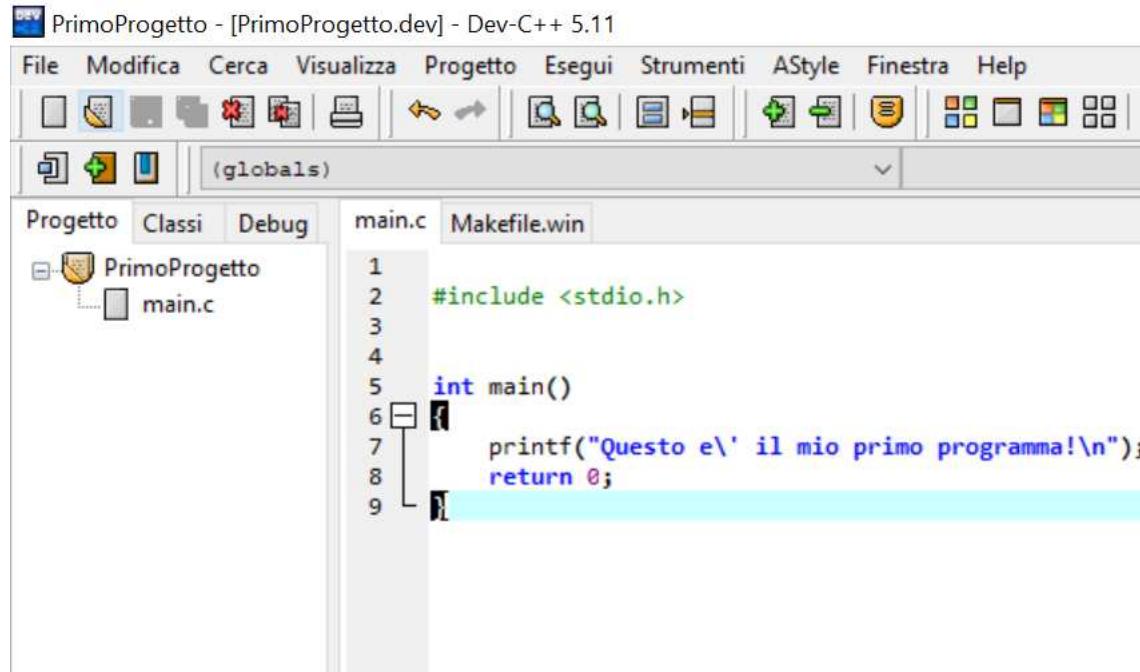


Salvataggio e recupero dei file

- È opportuno durante il lavoro salvare di tanto in tanto (utilizzando i comandi **Save Current File** o **Save All**) le modifiche apportate al programma onde prevenire malfunzionamenti o problemi vari che potrebbero compromettere il lavoro effettuato.
- I progetti o i files precedentemente salvati su disco possono essere recuperati utilizzando i comandi **Open Project** e **Open File**. Una volta aperto un progetto, i singoli files che lo compongono possono essere aperti cliccando sul loro nome nella area centrale a sinistra (**vista del progetto**) della finestra principale dell'ambiente di sviluppo.

Compilazione e linking di un progetto (1/4)

Consideriamo un semplice programma che stampa a schermo la riga di testo “*Questo è il mio primo programma!*”.

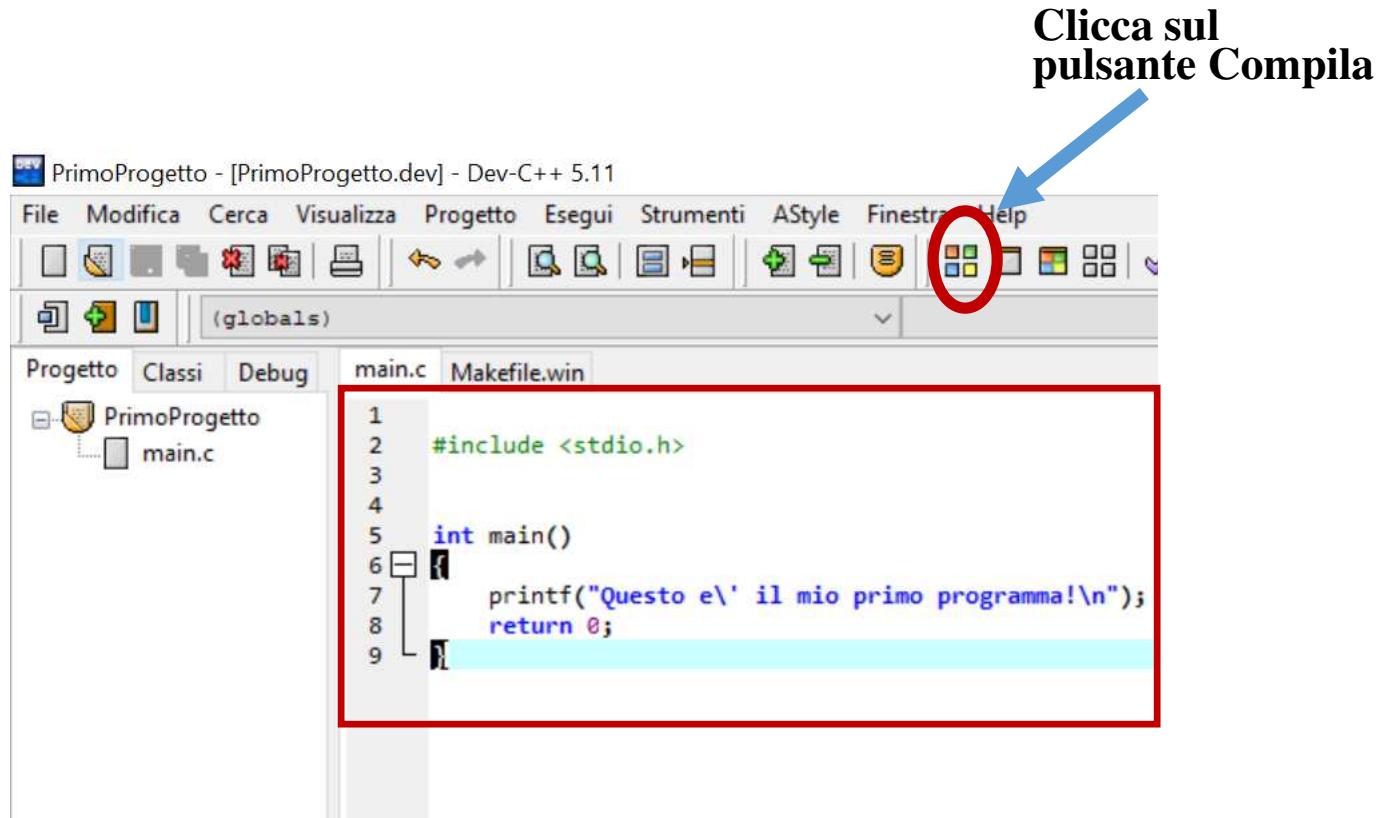


The screenshot shows the Dev-C++ 5.11 IDE interface. The title bar reads "PrimoProgetto - [PrimoProgetto.dev] - Dev-C++ 5.11". The menu bar includes File, Modifica, Cerca, Visualizza, Progetto, Esegui, Strumenti, AStyle, Finestra, and Help. The toolbar below has various icons for file operations like Open, Save, and Build. The left sidebar shows a project tree with "PrimoProgetto" expanded, containing "main.c". The main editor area has tabs for "main.c" and "Makefile.win", with "main.c" currently selected. The code in "main.c" is:

```
1 #include <stdio.h>
2
3
4
5 int main()
6 {
7     printf("Questo e' il mio primo programma!\n");
8     return 0;
9 }
```

Per eseguire la compilazione ed il linking utilizziamo il comando **Compila Progetto** attivabile o dalla barra dei menu o dalla barra degli strumenti.

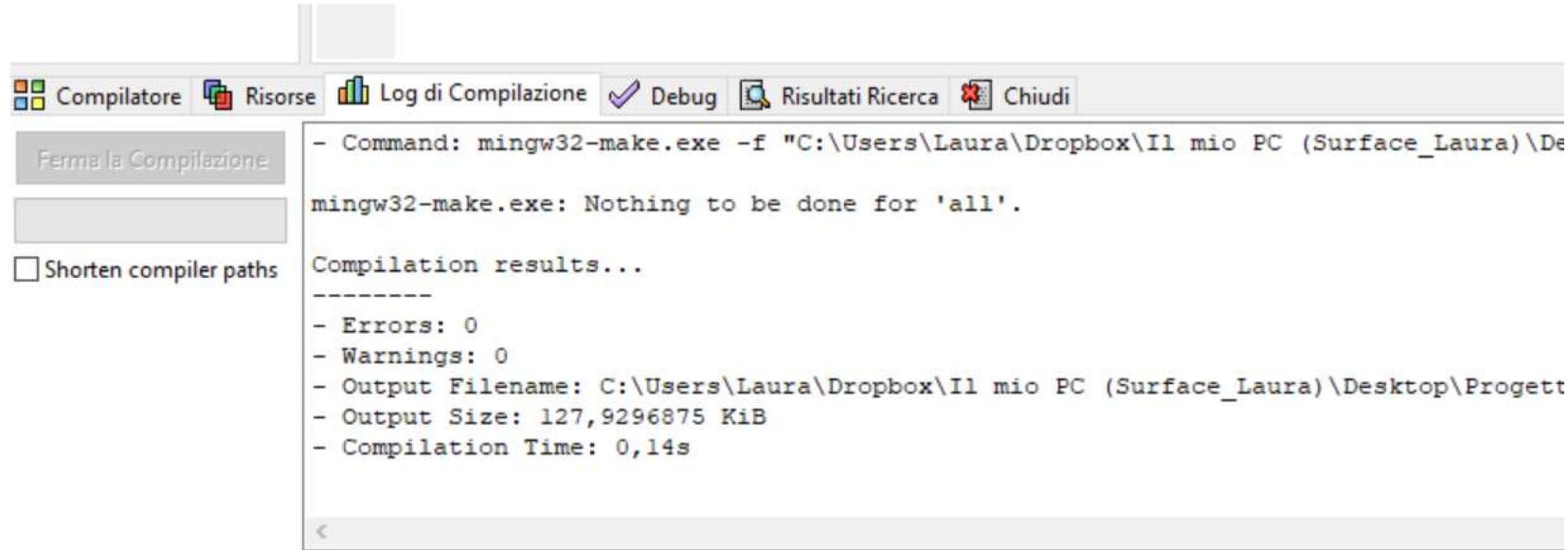
Compilazione e linking di un progetto (2/4)



Compilazione e linking di un progetto (3/4)

- Il programma è sintatticamente corretto e la compilazione va a buon fine. Viene creato il codice oggetto del file ‘**main.c**’ e salvato nel file ‘**main.o**’ della cartella del progetto.
- Viene effettuato il linking del file oggetto ‘**main.o**’ con il codice oggetto delle librerie standard del C.
- Sia la compilazione che il linking terminano con successo! Ciò è visualizzato nella sotto-finestra di output in basso dedicata alla compilazione e al linking (vedi slide seguente). Viene inoltre generato il file eseguibile ‘**main.exe**’ (memorizzato nella cartella del progetto).

Compilazione e linking di un progetto (4/4)



The screenshot shows a software window titled "Log di Compilazione" (Compilation Log). The window has several tabs at the top: "Compilatore" (Compiler), "Risorse" (Resources), "Log di Compilazione" (selected), "Debug", "Risultati Ricerca" (Search Results), and "Chiudi" (Close). Below the tabs, there is a button labeled "Ferma la Compilazione" (Stop Compilation) and a checkbox labeled "Shorten compiler paths". The main area displays the following text:

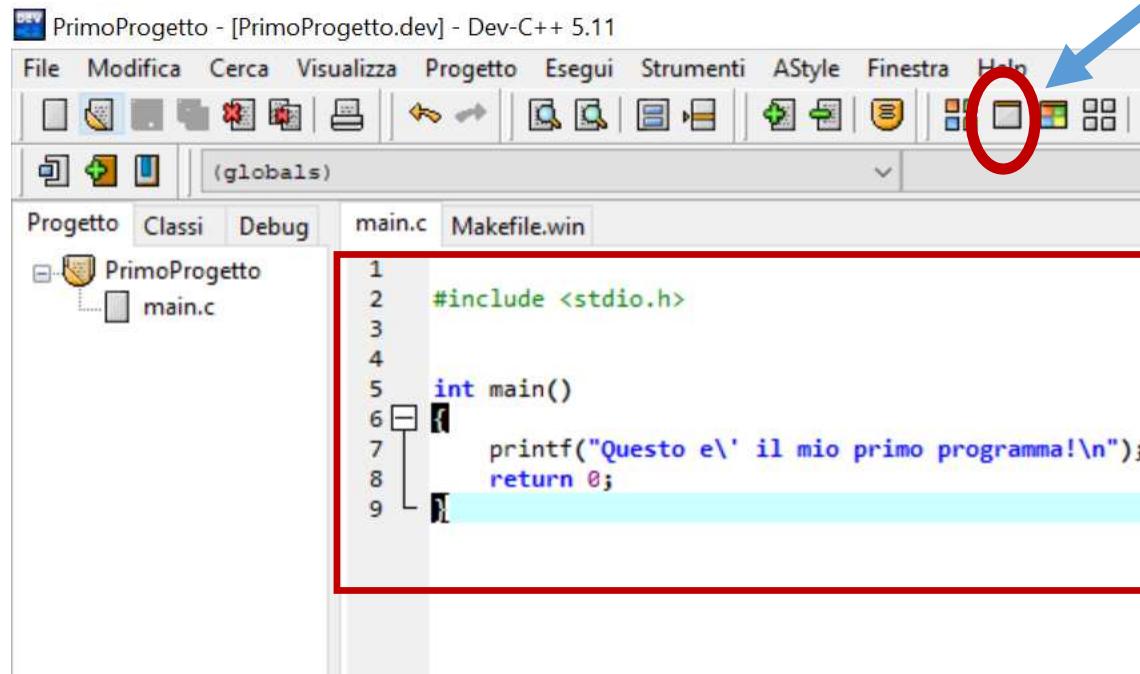
```
- Command: mingw32-make.exe -f "C:\Users\Laura\Dropbox\Il mio PC (Surface_Laura)\Desktop\Progetto\Makefile"
mingw32-make.exe: Nothing to be done for 'all'.

Compilation results...
-----
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Laura\Dropbox\Il mio PC (Surface_Laura)\Desktop\Progetto\main.exe
- Output Size: 127,9296875 KiB
- Compilation Time: 0,14s
```

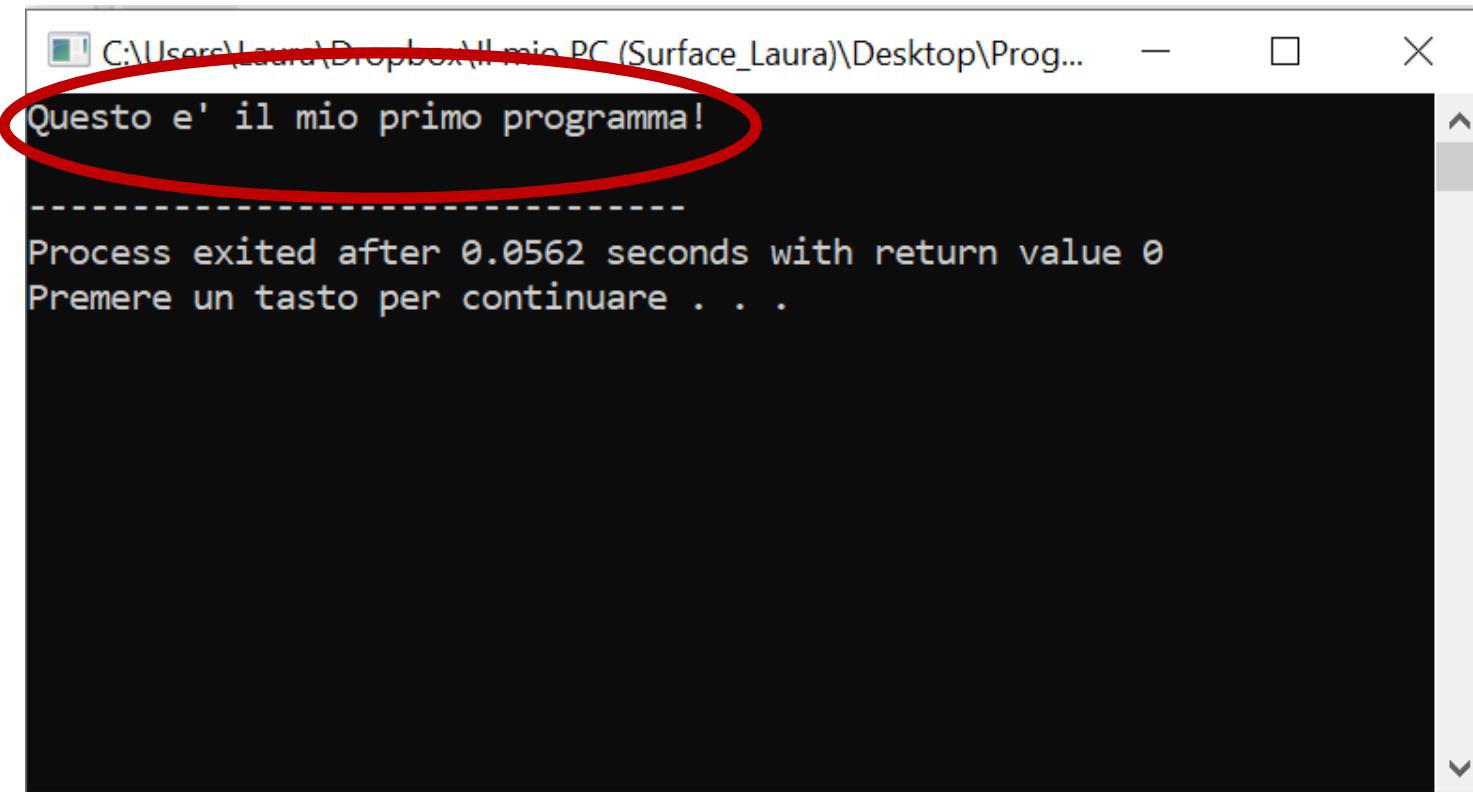
È possibile automaticamente eseguire il progetto (e, cioè, mandare in esecuzione il file eseguibile creato ‘**main.exe**’) utilizzando il comando **Esegui**.

Esecuzione di un progetto (1/2)

Clicka sul pulsante Esegui



Esecuzione di un progetto (2/2)



Viene visualizzata una finestra in ambiente MS-DOS (che consente solo interazione testuale) riportante l'output del programma e, cioè, la stringa di caratteri '*Questo è il mio primo programma!*'

Esame del programma esempio (1/4)

```
1 #include <stdio.h>
2
3
4
5 int main()
6 {
7     printf("Questo e\' il mio primo programma!\n");
8     return 0;
9 }
```

#include <stdio.h>

È un'istruzione per il preprocessore C ([direttiva del preprocessor](#)). Le direttive del preprocessor devono iniziare con il simbolo #.

La diretta **#include** deve essere eseguita dal nome di un [file di intestazione \(header\)](#) racchiuso tra parentesi angolari (< e >) se il file header è un file di sistema (libreria standard del C) e tra virgolette (es. *#include "esempio.h"*) per i **file header definiti dall'utente**. La diretta **#include** istruisce il preprocessore di sostituire la diretta con il contenuto dell'indicato file di intestazione.

Nel nostro caso, il file **stdio.h** è l'header della libreria di input/output standard contenente la dichiarazione del prototipo della funzione **printf**.

Esame del programma esempio (2/4)

```
int main()
```

```
{
```

Corpo

```
}
```

- Ogni programma in C deve contenere la definizione di una funzione particolare, chiamata **main**, che indica il punto di inizio del programma. Nel nostro caso, il **main** ha una lista vuota di parametri. La parola chiave **int** alla sinistra di **main** indica che **main** ‘restituisce’ un valore intero (numero intero).
- La porzione di testo **int main()** rappresenta l’intestazione della funzione **main**, l’interfaccia che essa fornisce all’esterno.
- Un **prototipo** di una funzione corrisponde alla sua intestazione e rappresenta l’unica informazione che uno deve avere per utilizzare la funzione tramite **istruzioni di chiamata a funzione**.

Esame del programma esempio (3/4)

```
int main()
{
    printf("Questo e\' il mio primo programma\n");
    return 0;
}
```

- La prima istruzione nel corpo del main è un'**istruzione di chiamata** alla funzione **printf** della libreria di input/output standard del C che prende in input la stringa di caratteri compresa tra doppi apici che verrà stampata sullo schermo.
- L'istruzione **return 0** comporta la terminazione della funzione **main** (il programma termina) che restituisce il valore 0. Il valore 0 indica che il programma termina con successo.
- Ogni istruzione nel corpo di una funzione deve terminare con il simbolo ; (chiamato **terminatore dell'istruzione**).

Esame del programma esempio (4/4)

```
int main()
{
    printf("Questo e\' il mio primo programma\n");
    return 0;
}
```

- Quando il compilatore compila l'istruzione di chiamata alla funzione **printf**, introduce semplicemente uno spazio nel codice oggetto **main.o** per una chiamata al tale funzione (parte da codificare). Compito del linker è di individuare il codice oggetto della libreria relativo alla funzione **printf** e sostituire la parte da codificare con opportuni istruzioni macchina.

Inserire commenti in un programma

```
1  /*
2   *      Description: Mio primo programma
3   */
4
5 #include <stdio.h>
6
7 //La funzione main inizial l'esecuzione del programma
8 int main()
9 {
10    printf("Questo e\' il mio primo programma!\n");
11    return 0;
12 } //fine della funzione main
```

- Commento a singola linea tramite il doppio slash //: ogni porzione di testo lungo una riga che è preceduta o inizia con // è considerata un commento.
- Commento a multiple linee tramite le sequenze di caratteri /* e */: ogni porzione di testo che inizia con /* e termina con */ è considerata un commento. Si noti che tale porzione di testo deve contenere solo una occorrenza di /* e similmente per */.
- I commenti sono *ignorati* dal compilatore e, dunque non provocano la generazione di alcun codice in linguaggio macchina.
- I commenti sono fondamentali per documentare e aumentare la leggibilità del codice!

Righe vuote e caratteri di spaziatura

```
1  /*
2   *      Description: Mio primo programma
3   */
4
5  #include <stdio.h>
6
7  //La funzione main inizial l'esecuzione del programma
8  int main()
9 {
10    printf("Questo e\' il mio primo programma!\n");
11    return 0;
12 } //fine della funzione main
```

Le righe 4 e 6 sono semplicemente righe vuote. È importante utilizzare righe vuote e i caratteri di spazio e tabulazione (**caratteri di spaziatura**) per rendere i programmi più leggibili. Tali caratteri sono ignorati dal compilatore.

Indentazione del corpo di una funzione

```
int main()
{
    printf("Questo e\' il mio primo programma!\n");
    return 0;
}
```

È altamente consigliato far rientrare a destra il testo dell'intero corpo di una funzione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano. Questa indentazione mette in risalto la struttura funzionale dei programmi e aiuta a renderli più leggibili.

Compilazione con errori (1/2)

```
1 #include <stdio.h>
2
3
4 int main()
5 {
6     printf("Questo e\' il mio primo programma!\n")
7     return 0;
8 }
9
```

- L'istruzione di chiamata alla funzione di libreria **printf** non termina con il simbolo ; (**errore sintattico**).
- La compilazione del programma termina con insuccesso. Il codice oggetto non viene generato e compare un messaggio di errore nella finestra di output dedicata al compilatore (vedi slide seguente) che fornisce utili indicazioni per rimuovere l'errore sintattico.

Compilazione con errori (2/2)

The screenshot shows a software interface for a compiler. At the top, there is a menu bar with tabs: 'Compilatore (3)', 'Risorse', 'Log di Compilazione', 'Debug', 'Risultati Ricerca', and 'Chiudi'. Below the menu bar is a table with three columns: 'Linea' (Line), 'Col...' (Column), 'Unità' (Unit), and 'Messaggio' (Message). The table contains the following data:

Linea	Col...	Unità	Messaggio
		C:\Users\Laura\Dropbox\Il mio PC (Surface_Laura)\Desktop\ProgettiLab\PrimoProgetto\main.c	In function 'main':
7	2	C:\Users\Laura\Dropbox\Il mio PC (Surface_Laura)\Desktop\ProgettiLab\PrimoProgetto\main.c	[Error] expected ';' before 'return'
28		C:\Users\Laura\Dropbox\Il mio PC (Surface_Laura)\Desktop\ProgettiLab\PrimoProgetto\Makefile.win	recipe for target 'main.o' failed

- Un doppio click sulla riga della finestra di output che spiega l'errore consente di evidenziare la riga del programma a cui si riferisce il messaggio di errore!

Linking con errori (1/2)

```
1 #include <stdio.h>
2
3
4 int main()
5 {
6     printf("Questo e\' il mio primo programma!\n")
7     return 0;
8 }
9
```

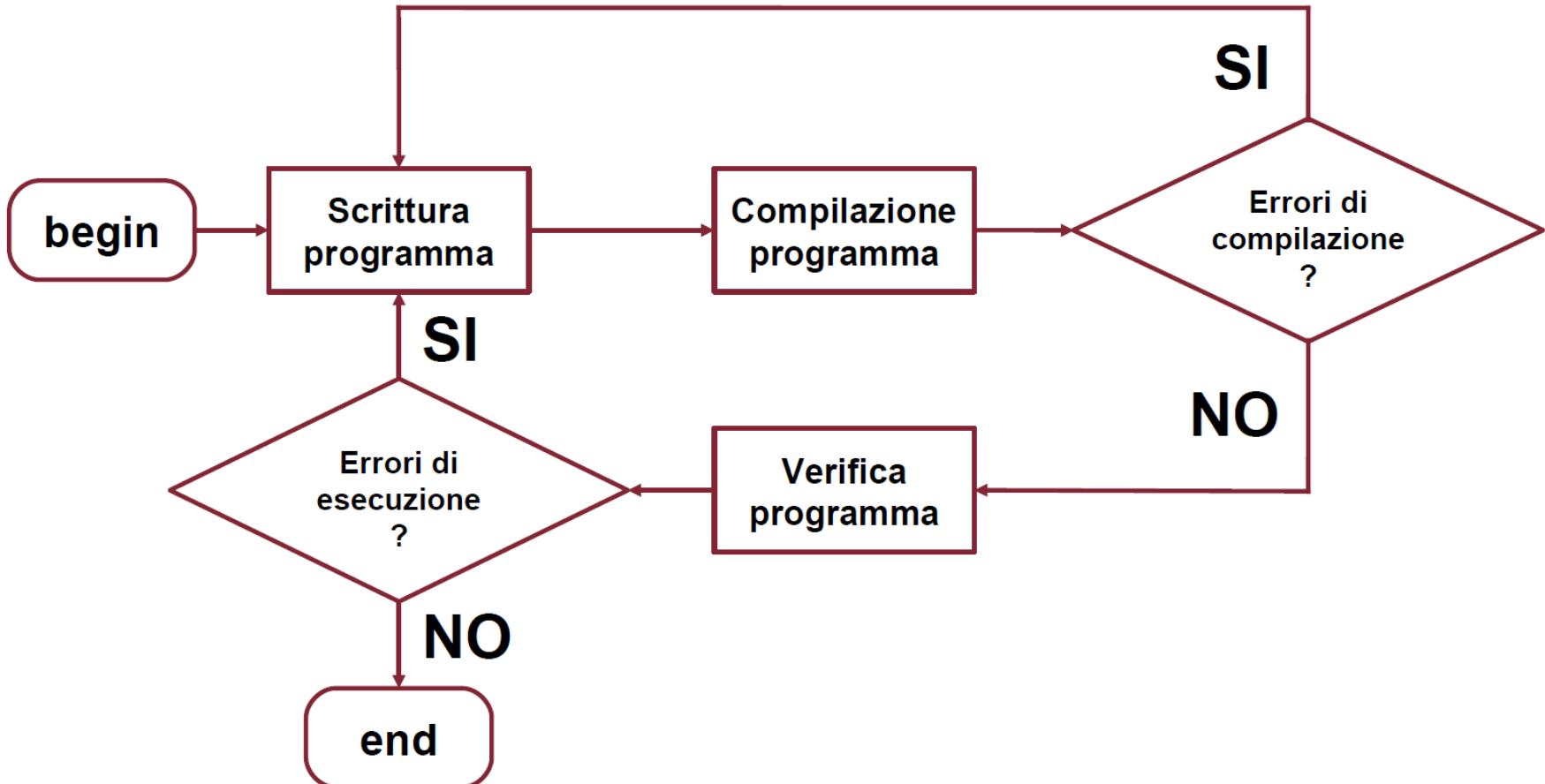
- L'istruzione di chiamata a funzione utilizza il nome **Printf** invece che **printf**. L'istruzione è sintatticamente corretta ed il compilatore non genera nessun errore.
- Comunque, né il programma né le librerie standard del C contengono tale definizione. Pertanto, il linker genera un errore (**errore semantico**). Il file eseguibile non viene generato e compare un messaggio di errore nella finestra di output dedicata al compilatore (vedi slide seguente) che fornisce indicazioni per rimuovere l'errore semantico. Si noti che l'errore è rilevato nel codice oggetto e non nel codice sorgente.

Il linguaggio C è case-sensitive: distingue tra lettere minuscole e maiuscole!

Linking con errori (2/2)

	Messaggio
\Desktop\ProgettiLab\PrimoProgetto\main.o	main.c:(.text+0x15): undefined reference to `Printf'
\Desktop\ProgettiLab\PrimoProgetto\collect2.exe	[Error] Id returned 1 exit status
\Desktop\ProgettiLab\PrimoProgetto\Makefile.win	recipe for target 'PrimoProgetto.exe' failed

Ciclo Edita-Compila-Verifica



Struttura dei file sorgente

- Un file sorgente in C ([file con esensione .c](#)) è strutturato come segue:
 - Eventuali istruzioni al preprocessore (direttive del preprocessore).
 - Eventuali dichiarazioni di variabili.
 - Eventuali definizioni di costanti.
 - Eventuali dichiarazioni di tipi definiti dall'utente.
 - Definizioni di funzioni.
- È buona norma che le direttive del preprocessore, se presenti, vengano poste all'inizio del file, seguite dalle eventuali dichiarazioni di variabili, definizioni di costanti, e dichiarazione dei tipi definiti dall'utente.
- **Ricordiamo che vi deve essere un unico file sorgente contenente la definizione della funzione main (punto di ingresso del programma).**

Struttura dei file di intestazione (header)

- Un file header in C ([file con esensione .h](#)) è strutturato come segue:
 - Eventuali istruzioni al preprocessore (direttive del preprocessore).
 - Eventuali dichiarazioni di variabili.
 - Eventuali definizioni di costanti.
 - Eventuali dichiarazioni di tipi definiti dall'utente.
 - Eventuali dichiarazioni di prototipi di funzioni.
- Si applicano le stesse regole di strutturazione del codice viste nella slide precedente.

Tipi semplici in C

- Tipi per numeri interi
- Tipi per numeri reali
- Tipi carattere

Per ciascun tipo consideriamo

- **Intervallo di definizione (che è sempre finito).**
- Notazione per specificare valori individuali (**costanti**).

Dato un tipo semplice T indicheremo con **sizeof(T)** il numero di byte usato nella sua rappresentazione binaria.

Tipi interi con segno

Tre tipi identificati dalle seguenti parole chiave:

- short
- int
- long

Intervallo di definizione (range): da -2^{N-1} a $2^{N-1} - 1$

N , multiplo di 8, è il numero di bit nella rappresentazione binaria e dipende dal compilatore.

- $\text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$
- $\text{sizeof(short)} \geq 2$ (ovvero almeno 2 byte =16 bit)
- $\text{sizeof(long)} \geq 4$ (ovvero almeno 4 byte =32 bit)

I valori limite sono contenuti nel file header **limits.h** della libreria standard che definisce le costanti:

- SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX

Tipi interi senza segno

Tre tipi per rappresentare interi non negativi. Identificati dalle parole chiave per i tipi interi con segno precedute dal qualificatore `unsigned`.

- `unsigned short`
- `unsigned int`
- `unsigned long`

Intervallo di definizione (range): da 0 a $2^N - 1$

N , multiplo di 8, corrisponde al numero di bit nella rappresentazione binaria del corrispondente tipo con segno.

I valori massimi (il valore minimo è sempre 0) sono sempre contenuti nel file header `limits.h` della libreria standard:

- `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`

Costanti per tipi interi (1/2)

Si possono utilizzare varie notazioni per specificare costanti intere (valori individuali di tipi interi):

- **Notazione decimale:** es. 1243, -3897. La prima cifra deve essere diversa da 0 (altrimenti, il compilatore la interpreta come una notazione ottale e, cioè, in base 8).
- **Notazione (decimale) esponenziale:** es. 12e+3 o 12E+3 rappresenta il numero 12×10^3 .
- **Notazione ottale:** sequenza di cifre tra 0 e 7 preceduta dalla cifra 0. Es. 01243, -00376. Ad esempio, la costante ottale 01243 denota l'intero:

$$1 \times 8^3 + 2 \times 8^2 + 4 \times 8^1 + 3 \times 8^0$$

- **Notazione esadecimale:** sequenza di caratteri consistenti nelle cifre decimali o nelle lettere da A a F (minuscole o maiuscole) preceduta dal prefisso 0x oppure dal prefisso 0X. Ad esempio, la costante esadecimale 0X1fB denota l'intero:

$$1 \times 16^2 + 15 \times 16^1 + 11 \times 16^0$$

Costanti per tipi interi (2/2)

Una costante intera specificata come descritto nella slide precedente può essere seguita da uno dei seguenti suffissi (**specificatori**):

- **L** (maiuscolo o minuscolo): la costante è interpretata come un intero di tipo **long**. Es. 1345L.
- **U** (maiuscolo o minuscolo): la costante è interpretata come un intero di tipo **unsigned**. Es. 1245U.
- **UL** o **LU** (con L e U maiuscole o minuscole): la costante è interpretata come un intero di tipo **unsigned long**. Es. 1245UL.

Tipi carattere

I tipi carattere in C sono tipi interi a 1 byte utilizzati per rappresentare i caratteri alfanumerici e alcuni caratteri speciali (**sequenze di escape**) in accordo allo standard di codifica dei caratteri ASCII (American Standard Code for Information Interchange) tramiti interi a 8 bits. Vi sono due tipi per caratteri in C, uno con segno e l'altro senza segno:

- `char`
- `unsigned char`

NOTA: dipendendo dal set di caratteri della macchina, i tipi carattere possono avere anche una rappresentazione a più byte. Vale comunque la regola che `sizeof(char) ≤ sizeof(int)`.

Tabella dei codici ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	:	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	□	127	7F	□

Costanti carattere

Una costante carattere è un simbolo di carattere racchiuso tra apici singoli come ‘a’ . Il valore di una costante carattere è il valore numerico di quel carattere all’interno del set di caratteri della macchina (usualmente il codice ASCII del carattere).

Alcuni caratteri speciali come il carattere *new line* (nuova riga) sono specificati come sequenze di due simboli (**sequenze di escape**) il cui primo simbolo è il carattere di backslash \. Ad esempio, il carattere new line è specificato tramite \n.

NOTA: alcune sequenze di escape sono utilizzate come comandi durante interazioni testuali in operazioni di input ed output. Ad esempio, il carattere *new line* consente di posizionare il **cursore (prompt) di lettura/scrittura** nella finestra di interazione all’inizio della riga successiva a quella corrente.

Elenco completo delle sequenze di escape

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

Tipi reali

Tre tipi identificati dalle seguenti parole chiave:

- **float** (singola precisione)
- **double** (doppia precisione)
- **long double** (doppia precisione)

Intervallo di definizione (range): finito e basato su rappresentazione a virgola mobile.

- `sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)`
- `sizeof(float) ≥ 4`

I valori limite sono contenuti nel file header **float.h** della libreria standard che definisce le costanti:

- `FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX, LDBL_MIN, LDBL_MAX`

Costanti reali

Si possono utilizzare due notazioni per specificare costanti reali (valori individuali di tipi reali):

- Notazione decimale con punto decimale: es. 1.243, -389.74.
- Notazione (decimale) esponenziale: es. 12.56e+3 o 12.56E+3 rappresenta il numero 12.56×10^3 .

Una costante reale può essere seguita da uno dei seguenti suffissi (**specificatori**):

- **F** (maiuscolo o minuscolo): la costante è interpretata come un numero reale di tipo **float**. Es. 13.45f.
- **L** (maiuscolo o minuscolo): la costante è interpretata come un numero reale di tipo **long double**. Es. 12.45L.

Tipi reali: rappresentazione a virgola mobile

I valori che una variabile reale a virgola mobile può assumere sono descritti in termini di

- **Precisione:** il numero di cifre decimali significative che possono essere rappresentate.
- **Range:** la più grande e più piccola parte integrale che una variabile di quel tipo può assumere.

Nel sistema decimale, un valore reale non-negativo di uno specifico tipo (**float**, **double**, **long double**) viene rappresentato come

$$0.d_1 \dots d_P \times 10^N$$

dove $d_1 \neq 0$, P è il massimo numero di cifre decimali significative (**precisione**), e N è compreso tra -R e R, dove R (intero positivo) definisce univocamente il **range**.

Imprecisioni nei calcoli con numeri reali

- I tipi reali possono rappresentare solo un numero finito di numeri razionali aventi un numero limitato di cifre decimali. Gli altri valori nello specifico range (vedi slide precedente) sono approssimati.
- A differenza dei tipi interi, le operazioni sui reali introducono errori di precisione che si propagano amplificandosi in calcoli complessi. È un problema tipico, ad esempio, nell'Analisi Numerica.

Lezione 4: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi

1. Scrivere un programma che, dati due interi (inseriti da tastiera), li confronti stampando a video se i due interi sono uno più grande dell'altro, uno più piccolo dell'altro oppure sono uguali.
2. Si scriva un programma capace di compiere le 4 operazioni (somma, sottrazione, moltiplicazione e divisione) tra due numeri reali inseriti da tastiera. Dopo che sono stati inseriti i due numeri, detti A e B, il programma dovrà stampare a video i quattro valori $A+B$, $A-B$, $A*B$, A/B , su due righe diverse (due per riga) . Si ipotizzi che sia B diverso da 0.
3. Scrivere un programma che date le età di tre persone, immesse da tastiera e supposte rappresentate con numeri interi, calcoli l'età media delle persone (somma delle età diviso 3), e stamparla a video.

Esercizi

4. Scrivere un programma che, dato un intero in input, determini se esso è pari o dispari e stampi tale informazione a video.
5. Scrivere un programma che, dati due interi (inseriti da tastiera), stabilisca se uno è multiplo dell'altro e stampi tale informazione a video.
6. Scrivere un programma che, dati tre numeri da tastiera, li stampi in ordine crescente.
7. Scrivere un programma che prenda in input 10 valori da tastiera, e che per ogni valore inserito, stampi la media di tutti i numeri già inseriti.
8. Scrivere un programma che prenda in input un numero intero da tastiera, e stampi le ultime due cifre decimali su righe separate. Ad esempio, per il numero 1256, si dovrà stampare prima 5 e poi 6.

Lezione 8: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi

1. Scrivere un programma che dato un intero in input (inserito da tastiera), stampi il cubo dell'intero utilizzando solo operazioni di somma.
2. Si scriva un programma che dato un intero N in input (inserito da tastiera) stampi a video i **fattori primi** di N (e, cioè, i divisori di N che sono numeri primi).
3. Un numero intero positivo è **triangolare** se coincide con la somma dei primi N numeri positivi per qualche N. Scrivere un programma che dato in input un intero positivo (inserito da tastiera) verifichi se il numero è triangolare o meno e stampi a video un opportuno messaggio che nel caso in cui il numero sia triangolare fornisca il valore N tale che $1+2+\dots+N$ coincida con il numero dato.

Esercizi

4. Scrivere un programma che dato in input un intero positivo $N \leq 100$ stampi i primi N interi positivi **altamente composti**. Un intero positivo è altamente composto se qualunque intero positivo minore di esso ha meno divisori. I primi 20 numeri altamente composti sono 1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, 1260, 1680, 2520, 5040, 7560.
5. Su una scacchiera 8x8 sono posizionati due pezzi: il Re bianco e la Regina nera. Si scriva un programma che, acquisite le posizioni del Re e della Regina, determini se la Regina è in posizione tale da poter mangiare il Re. Le posizioni dei due pezzi sono identificate dalla riga e la colonna su cui si trovano, espresse come numeri interi tra 1 e 8.
6. Scrivere un programma che legge un intero positivo N da tastiera e verifica se N può essere scomposto nella somma dei quadrati di due interi. In caso positivo, stampare una possibile decomposizione e cioè due interi tali che la somma dei loro quadrati coincida con N .

Esercizi

4. Scrivere un programma che dato in input un intero positivo $N \leq 100$ stampi i primi N interi positivi **altamente composti**. Un intero positivo è altamente composto se qualunque intero positivo minore di esso ha meno divisori. I primi 20 numeri altamente composti sono 1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, 1260, 1680, 2520, 5040, 7560.
5. Su una scacchiera 8x8 sono posizionati due pezzi: il Re bianco e la Regina nera. Si scriva un programma che, acquisite le posizioni del Re e della Regina, determini se la Regina è in posizione tale da poter mangiare il Re. Le posizioni dei due pezzi sono identificate dalla riga e la colonna su cui si trovano, espresse come numeri interi tra 1 e 8.
6. Scrivere un programma che legge un intero positivo N da tastiera e verifica se N può essere scomposto nella somma dei quadrati di due interi. In caso positivo, stampare una possibile decomposizione e cioè due interi tali che la somma dei loro quadrati coincida con N .

Esercizi

7. Scrivere un programma che legge da tastiera una sequenza (di lunghezza arbitraria) di numeri interi positivi, terminata da 0 (sentinella), e indica, alla fine della sequenza, qual è la lunghezza della massima sottosequenza di numeri consecutivi in ordine crescente.

Esempi:

13 3 → 8 4 → 5 1 → 17 0
Lung. max = 2

21 19 18 14 9 6 4 3 0
Lung. max = 1

2 1 → 3 → 6 → 8 5 1 → 12 → 18 17 0
Lung. max = 4

Lezione 9

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 9: Funzioni e ricorsione

- Funzioni in C.
- Definizioni e prototipi di funzione.
- Chiamate di funzione e passaggio di argomenti per valore.
- Restituzione di controllo al chiamante.
- Variabili globali e locali.
- Spazio dei nomi e campo di azione delle funzioni e variabili.
- Gestione dello stack per la chiamata e ritorno da funzioni.
- Approccio ricorsivo alla risoluzione di problemi.
- Funzioni ricorsive.

Struttura di un programma in C

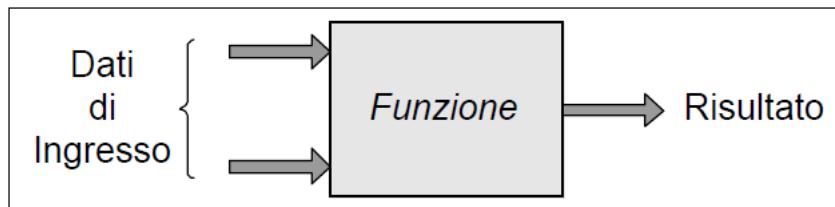
- Un programma in C consiste essenzialmente di un insieme di **definizioni di funzioni e definizioni di variabili globali**.
- All'interno di un file sorgente .c le **definizioni di funzioni e le definizioni di variabili globali** possono essere disposte in un ordine qualsiasi.
- Un programma in C può essere suddiviso in diversi file sorgente. La definizione di una singola funzione deve trovarsi in unico file sorgente.

Funzioni in C (1/5)

- Una funzione realizza un compito specifico (algoritmo) che può essere utilizzato in diversi punti del programma.
- Una funzione “raggruppa” una sequenza di istruzioni sotto un unico nome (**unità di calcolo**). Le particolari istruzioni che definiscono una funzione sono scritte una sola volta e **sono nascoste** alle altre funzioni.
- Una funzione viene utilizzata (invocata) tramite il meccanismo di **chiamata di funzione**. Una chiamata di funzione specifica il nome della funzione e fornisce le informazioni (come **argomenti**) di cui la funzione ha bisogno per eseguire il suo compito designato.

Funzioni in C (2/5)

- Idealmente, una funzione prende dati in ingresso, esegue istruzioni che operano sui dati in ingresso, e restituisce un risultato in uscita.



- Una funzione in linguaggio C ha un **nome** (identificativo), un **elenco (possibilmente vuoto) di parametri di ingresso**, un **corpo** (sequenza di istruzioni dove i parametri di ingresso vengono utilizzati per produrre eventualmente un **risultato in uscita**), ed un **eventuale risultato di uscita**.

Funzioni in C (3/5)

- Ogni programma in C deve contenere una funzione particolare, chiamata **main**, che indica il punto di inizio del programma (la prima istruzione della funzione è la prima eseguita nel programma) ed i cui parametri vengono specificati dall'utente quando fa partire il programma.
- Nel corpo della funzione **main** vengono richiamate altre funzioni, che a loro volta possono usarne altre ancora, e così via.
- Oltre alle funzioni definite direttamente nel programma, è possibile utilizzare funzioni di librerie già compilate, ed, in particolare, le funzioni della **libreria standard del C**.
- La comunicazione tra le funzioni avviene tramite gli argomenti, i valori di ritorno delle funzioni stesse e le variabili globali.

Funzioni in C (4/5)

Le funzioni facilitano la progettazione, l'implementazione, l'utilizzo e la manutenzione di programmi di grosse dimensioni.

- **Approccio *dividi et impera*:** il programma è suddiviso in sottoprogrammi più semplici corrispondenti a singole definizioni di funzioni. Le istruzioni associate ad una singola funzione sono nascoste alle altre funzioni. Ciò aiuta a focalizzare l'attenzione su come risolvere il problema senza scendere nei dettagli implementativi. Le funzioni rendono più chiaro il programma nel suo complesso e ne facilitano la manutenzione.
- **Codice succinto:** le funzioni consentono di evitare di ripetere una porzione di codice che realizza un certo compito. Impacchettare il codice come funzione ne consente l'esecuzione in altri punti del programma semplicemente chiamando la funzione.

Funzioni in C (5/5)

Le funzioni facilitano la progettazione, l'implementazione, l'utilizzo e la manutenzione di programmi di grosse dimensioni.

- **Modularità**: una funzione generalmente è un'unità di calcolo indipendente dal contesto o ambiente (programma) in cui viene utilizzata.
- **Riusabilità del software**: data la caratteristica di indipendenza, ogni funzione può essere implementata in modo tale da poter essere riutilizzata in programmi differenti. Si possono usare funzioni esistenti come **blocchi costituenti** per creare nuovi programmi.

Costrutti per le funzioni in C

- **Definizione di funzione:** realizzazione di una specifica funzione. Ad ogni funzione di un programma è associata un'unica definizione, scritta in qualche file sorgente del programma.
- **Dichiarazione o prototipo di funzione:** rappresenta l'interfaccia di una funzione, l'informazione che una funzione mette a disposizione all'ambiente (programma) per consentirne l'utilizzo. Una dichiarazione di funzione include il nome della funzione, il tipo di dato restituito, e l'elenco dei parametri formali d'input.
- **Chiamata di funzione:** meccanismo attraverso il quale una funzione può essere utilizzata nel corpo di istruzioni di un'altra funzione specificando il nome della funzione e i dati per i parametri d'input (**argomenti o parametri attuali nella chiamata di funzione**).

Sintassi definizione di funzione (1/4)

Sintassi

```
<tipo valore ritorno> <nome funzione>(<elenco parametri>)
```

```
{
```

```
    <corpo>
```

```
}
```

- La parte che precede la parentesi graffa { rappresenta l'**intestazione della funzione**.
- **<tipo valore ritorno>**: è il tipo del risultato restituito.
- **<nome funzione>**: è qualsiasi identificatore valido (come visto per gli identificatori delle variabili semplici).
- **<elenco parametri>**: elenco separato da virgole che specifica i parametri che la funzione riceve quando è chiamata.
- **<corpo>**: rappresenta una sequenza di istruzioni (istruzioni atomiche o di controllo) che include dichiarazioni di variabili.

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione di iterazione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Sintassi definizione di funzione (2/4)

Sintassi

```
<tipo valore ritorno> <nome funzione>(<elenco parametri>)
{
    <corpo>
}
```

Tipo del valore di ritorno <tipo valore ritorno>:

- Per dati semplici e, cioè, numerici rappresenta la parola chiave o la sequenza di parole chiave per identificare un tipo predefinito numerico. Es: **char, short, int, long, float, double, long double**.
- Per tipi di dati definiti dall'utente, è l'identificatore del tipo di dato.
- Si utilizza la parola chiave **void** per indicare che la funzione non restituisce alcun valore.

Esempio:

```
void StampaSomma(int x,int y)
{
    printf("La somma e\' %d", x+y);
}
```

Sintassi definizione di funzione (3/4)

Sintassi

```
<tipo valore ritorno> <nome funzione>(<elenco parametri>)
{
    <corpo>
}
```

Elenco dei parametri **<elenco parametri>**: sequenza, possibilmente vuota, di **dichiarazioni di parametri formali** separate dalla virgola.

Sintatticamente una dichiarazione di parametro formale corrisponde

- o ad una dichiarazione di variabile senza inizializzazione e senza il terminatore di istruzione ;.
- o ad una dichiarazione di funzione (parametri funzionali) senza il terminatore di istruzione ; : aspetto avanzato che vedremo dopo.

Buona norma: scegliere nomi significativi per le funzioni e nomi significativi per i parametri formali (identificatori nelle dichiarazioni dei parametri formali) rende i programmi più leggibili e contribuisce a evitare un uso eccessivo di commenti.

Sintassi definizione di funzione (4/4)

Sintassi

```
<tipo valore ritorno> <nome funzione>(<elenco parametri>)
{
    <corpo>
}
```

Il corpo di istruzioni **<corpo>** non può contenere definizioni di funzioni (definizioni di funzioni non possono essere annidate) ma può contenere dichiarazioni di funzioni (prototipi) per poter utilizzare (tramite chiamata a funzione) funzioni definite altrove e non “visibili” dalla funzione in questione.

Esempio:

```
void Prova(int c)
{
    int CalcolaQuadrato(int); //funzione definita altrove
    printf("Il quadrato di %d e' %d\n", c, CalcolaQuadrato(c));
}
```

Dichiarazione o prototipo di funzione (1/2)

Sintassi

Corrisponde all'intestazione della corrispondente definizione di funzione seguita dal terminatore di istruzione ; . Nella lista dei parametri formali è possibile utilizzare nomi diversi rispetto a quelli utilizzati nella definizione di funzione, ed è anche possibile omettere i nomi.

Esempio:

```
int CalcolaQuadrato(int num)
{
    return num*num;
}
```

```
int CalcolaQuadrato(int);
```

Definizione funzione

Dichiarazione funzione

Dichiarazione o prototipo di funzione (2/2)

- Può comparire all'esterno delle definizioni di funzioni o come parte del corpo di istruzioni di una definizione di funzione.
- Una dichiarazione di funzione è utilizzata per estendere il **campo di azione o scope dell'identificatore** della funzione e cioè la porzione di programma all'interno della quale la funzione può essere utilizzata (tramite chiamate a funzione).
- Il campo di azione di una definizione di funzione o di una dichiarazione di funzione va dal punto in cui tale definizione o dichiarazione è scritta al termine del file sorgente in cui si trova.
- **Buona norma:** per utilizzare una funzione definita in un altro file sorgente è buona norma specificare il prototipo della funzione in un file header .h ed includere tale file nel file sorgente tramite la direttiva **#include** del preprocessore.

Sintassi chiamata di funzione (1/3)

Sintassi

<nome funzione> (<elenco argomenti>)

Elenco dei argomenti o parametri attuali <elenco argomenti>: sequenza, possibilmente vuota, di **argomenti** (detti anche **espressioni di assegnazione**).

- Il numero di argomenti deve corrispondere al numero (eventualmente nullo) di parametri formali nella definizione di funzione.
- Ogni argomento deve essere di tipo compatibile con il tipo del corrispondente parametro formale.

Sintassi chiamata di funzione (2/3)

Sintassi

<nome funzione> (<elenco argomenti>)

- Per un parametro formale di **tipo numerico** (parametro semplice), un argomento (o parametro attuale) può essere una generica espressione numerica.
- Per un parametro formale di tipo definito da utente, un argomento deve essere dello stesso tipo (ad esempio una variabile dello stesso tipo).
- Per un parametro funzionale, un argomento deve essere il nome di una funzione avente lo stesso tipo di ritorno e la stessa lista di parametri (a meno dei nomi dei parametri) della dichiarazione del parametro funzionale. **NOTA: una dichiarazione di parametro funzionale non può corrispondere ad una dichiarazione di funzione con parametri funzionali.**

Sintassi chiamata di funzione (3/3)

Sintassi

<nome funzione> (<elenco argomenti>)

Esempio:

```
static int square(int x)
{
    return x*x;
}

int Prova(int c,int f(int))
{
    return c+f(7);
}

int main()
{
    float x = 5.45;
    float risultato = Prova(7,square) * x;
    printf("\n Il risultato e' %f\n",risultato);
    return 0;
}
```

Semantica della chiamata di funzione (1/2)

<nome funzione> (<elenco argomenti>)

Semantica

- I parametri formali non-funzionali nella definizione di funzione sono gestiti come tutte le altre variabili dichiarate nel corpo della funzione.
- Nell'esecuzione della chiamata, ciascun parametro formale è inizializzato con il valore del corrispondente argomento (parametro attuale).
- Al termine dell'inizializzazione dei parametri formali viene eseguito il corpo della funzione (che in generale può accedere in lettura o in scrittura ai parametri formali).
- Al termine dell'esecuzione del corpo della funzione, il controllo ritorna alla funzione chiamante.
- L'istruzione **return** nel corpo di una funzione consente di restituire il controllo al contesto chiamante.

Semantica della chiamata di funzione (2/2)

<nome funzione> (<elenco argomenti>)

Semantica

Per ogni parametro formale semplice (di tipo numerico), il corrispondente argomento deve essere di tipo numerico.

- Nell'inizializzazione del parametro formale, il valore dell'argomento viene convertito nel tipo del valore del parametro formale in accordo a quanto visto per le conversioni implicite di tipo numerico per gli operatori di assegnazione.
- È possibile che un valore di tipo "superiore" venga convertito in un valore di tipo "inferiore" con possibile perdita di informazione ed errori di **overflow**.
 - Per interi, i bit più significativi in eccesso vengono rimossi.
 - Dai reali ad interi, si ha il troncamento della parte frazionaria.

Utilizzo di una chiamata di funzione

Le chiamate di funzione sono specificate all'interno del corpo di una funzione come:

- istruzione di chiamata di funzione: chiamata di funzione seguita dal terminatore di istruzione ; . In questo caso, anche se la funzione restituisce un risultato (il tipo del valore di ritorno non è **void**), tale risultato non viene utilizzato.
- Come sotto-espressioni numeriche per funzioni che restituiscono dati numerici.
- Per formare sotto-espressioni numeriche per funzioni che restituiscono puntatori a dati numerici (lo vedremo in seguito).
- Come operandi destri di operatori di assegnazione per variabili di tipo aggregato (strutture o unioni) che vedremo in seguito.

Passaggio argomenti per valore e per riferimento

In molti linguaggi di programmazione ci sono due modi per passare gli argomenti rappresentati da variabili in una chiamata di funzione: il **passaggio per valore** e il **passaggio per riferimento**.

- **Passaggio per valore:** il parametro formale corrisponde ad una *copia* dell'argomento. Esso viene inizializzato con il valore assunto dall'argomento. Le modifiche alla copia **non incidono** sul valore della variabile originaria nella funzione chiamante.
- **Passaggio per riferimento:** quando un argomento rappresentato da una variabile è passato per riferimento, la funzione chiamata può modificare il valore della variabile originaria. Il parametro formale "punta" alla stessa locazione di memoria della variabile originaria.

Passaggio argomenti per valore e per riferimento

- Il **Passaggio per valore** va usato ogni volta che la funzione chiamata non ha necessità di modificare il valore della variabile originaria della funzione chiamante. Questo evita **effetti collaterali** (modifica delle variabili) che sono di grande impedimento allo sviluppo di sistemi software corretti e affidabili.
- Nel linguaggio C, tutti gli argomenti sono passati per valore. È possibile ottenere implicitamente il passaggio per riferimento utilizzando come argomento **un puntatore** così che la funzione chiamata possa cambiare il valore dell'oggetto puntato (lo vedremo in seguito).

Restituzione del controllo da una funzione (1/2)

Vi sono tre modi per restituire il controllo da una funzione chiamata al punto in cui tale funzione è stata invocata.

1. Tramite l'istruzione **return**; nel corpo della funzione chiamata. Nel caso in cui la funzione ha un tipo di valore di ritorno (tipo diverso da **void**), il valore di ritorno è "indefinito" (cioè non si può fare nessuna assunzione sul valore restituito).

2. Per funzioni che restituiscono un valore, tramite l'istruzione **return <espressione>**; dove per tipi numerici di valori di ritorno, **<espressione>** è una generica espressione numerica. In questo caso, il valore dell'espressione viene convertito nel tipo del valore di ritorno in accordo a quanto visto per le conversioni implicite di tipo numerico per gli operatori di assegnazione.

Restituzione del controllo da una funzione (2/2)

3. Un altro caso in cui il chiamante riprende il controllo dell'esecuzione, senza che gli venga passato esplicitamente alcuno valore di ritorno, è quello in cui l'esecuzione del corpo della funzione chiamata termina per il raggiungimento della parentesi graffa di chiusura.

NOTA: per funzioni che hanno un tipo di ritorno non **void**, i casi 1 e 3 sono consentiti, ma il fatto che una funzione, in un punto, ritorni un valore ed in un altro non ritorni nulla, è indice di una situazione anomala che dovrebbe essere impedita.

Spazio dei nomi per funzioni (1/3)

- Abbiamo visto che il **campo di azione o scope** di una definizione di funzione (e cioè la porzione di programma all'interno della quale la funzione può essere utilizzata) va dal punto in cui tale definizione è scritta al termine del file sorgente in cui si trova.
- Per estendere il campo di azione di una definizione di funzione basta specificare il prototipo (dichiarazione) della funzione in ogni file in cui la funzione è invocata. Il campo di azione di un prototipo si estende dal punto in cui è specificata la dichiarazione fino alla fine del file sorgente. Un prototipo di una funzione può essere specificato anche all'interno del corpo di una funzione.
- I prototipi di funzione indicano al compilatore che la funzione specificata è definita o in seguito nello stesso file o in un file differente. Il compilatore non tenta di risolvere i riferimenti a una tale funzione: questo compito è lasciato al linker. Se il linker non riesce a localizzare un'adeguata definizione della funzione, emette un messaggio di errore.

Spazio dei nomi per funzioni (2/3)

- Le funzioni per **default** hanno un **collegamento esterno**: è possibile accedervi in altri file che contengano i corrispondenti prototipi di funzione.
- Per cambiare il comportamento di default (collegamento esterno), si utilizza lo *specificatore della classe di memoria static*, premettendo la parola chiave **static** nell'intestazione della definizione di funzione (**collegamento interno**). Ciò impedisce che la funzione possa essere utilizzata da una funzione che non è definita nello stesso file.
- Lo specificatore di classe di memoria **static** consente di tenere nascoste all'interno di un file sorgente funzioni (o anche variabili globali) utilizzate per calcoli intermedi da parte di altre funzioni definite all'interno dello stesso file.

Esempio:

```
static int square(int x)
{
    return x*x;
}
```

Spazio dei nomi per funzioni (3/3)

- Dalle regole sul campo d'azione, non è possibile definire nello stesso file due funzioni con lo stesso nome (identificatore).
- Inoltre, il C non consente due definizioni di funzioni con collegamento esterno (senza la parola chiave **static**) aventi lo stesso identificatore (nome della funzione), anche se le due funzioni sono definite in files sorgenti diversi.
- Per avere multiple definizioni di funzioni con lo stesso nome, tali definizioni devono risiedere in file sorgenti distinti e tutte ad eccezione di una devono essere definite con lo specificatore di classe di memoria **static**.

Variabili globali (1/2)

- Le variabili globali, dette anche **variabili esterne**, sono dichiarate all'esterno delle definizioni di funzione.
- Le funzioni e le variabili globali hanno un **campo di memoria statico**. Ciò significa che un'area della memoria principale è allocata per memorizzare una variabile globale (risp., una funzione) all'inizio dell'esecuzione del programma, e tale area di memoria rimane associata alla variabile (risp., funzione) per l'intera esecuzione del programma.
- Lo spazio di memoria di una variabile viene allocato ed inizializzato per default a zero, a meno che sia diversamente inizializzato dal programmatore.

Variabili globali (2/2)

Le variabili globali forniscono un modo alternativo di comunicare dati tra le funzioni, diverso dal meccanismo di chiamata di funzione.

Qualsiasi funzione può accedere ad una variabile globale attraverso il suo nome (identificatore) a patto che tale nome sia visibile nel punto di utilizzo. È bene usare poche variabili globali perché:

- Restano in vita sempre, anche se non servono più.
- Possono essere cambiate in modo inaspettato da più funzioni.
- Possono distruggere la generalità di certe funzioni, legandole ai nomi delle variabili globali che esse manipolano.

Spazio dei nomi per variabili globali (1/3)

- Il **campo di azione o scope** di una variabile (variabile globale o non) rappresenta la porzione di programma all'interno della quale l'identificatore della variabile può essere utilizzato.
- Similmente alle funzioni, la definizione (alias dichiarazione) di una variabile globale ha un campo d'azione **esteso al file**, e, cioè, si estende dal punto in cui è specificata la dichiarazione fino alla fine del file sorgente.
- Per estendere il campo di azione di una definizione di variabile globale, si utilizza un costrutto simile concettualmente al prototipo di funzione chiamata **dichiarazione extern**. Una dichiarazione extern per una variabile globale corrisponde alla dichiarazione della variabile senza inizializzazione preceduta dallo *specificatore della classe di memoria extern*. Essa può essere utilizzata anche all'interno del corpo di una funzione.

Esempio:

`extern int contatore;`

Spazio dei nomi per variabili globali (2/3)

- Una **dichiarazione extern** per una variabile globale non alloca nessuna memoria ma indica al compilatore che la variabile globale è definita o in seguito nello stesso file o in un file differente. Il compilatore non tenta di risolvere i riferimenti a tale variabile: questo compito è lasciato al linker. Se il linker non riesce a localizzare un'adeguata definizione della variabile globale, emette un messaggio di errore.
- Similmente alle funzioni, le variabili globali hanno per **default** un **collegamento esterno**: è possibile accedervi in altri file che contengano le corrispondenti dichiarazioni extern.
- Per cambiare il comportamento di default (collegamento esterno), si utilizza lo *specificatore della classe di memoria static*, premettendo la parola chiave **static** nella definizione della variabile globale. Ciò impedisce che la variabile possa essere utilizzata da una funzione che non è definita nello stesso file.

Spazio dei nomi per variabili globali (3/3)

- Dalle regole sul campo d'azione, non è possibile definire nello stesso file funzioni o variabili globali con lo stesso nome (identificatore).
- Comunque a differenza delle funzioni, sono consentite definizioni di variabili globali (senza la parola chiave **static**) aventi lo stesso identificatore in file sorgenti distinti. Ciò può creare ambiguità. È buona norma dunque che gli identificatori di variabili globali definite senza lo specificatore static siano distinti.

Variabili definite in una funzione (1/2)

- Le variabili definite in una funzione sono o i parametri formali nella definizione della funzione o le variabili definite nel corpo della funzione (dette anche **variabili locali**).
- Le variabili corrispondenti ai parametri formali di una funzione hanno un **campo di memoria automatico**. Ciò significa che tali variabili vengono riallocate (create) ogni volta che una funzione viene eseguita (tramite chiamata di funzione), esistono finchè l'esecuzione della funzione è attiva e sono distrutte quando la funzione restituisce il controllo al chiamante.
- Per **default**, le variabili locali sono gestite come i parametri formali e, cioè, hanno un campo di memoria automatico. Dunque, i loro valori non sono preservati tra chiamate consecutive alla funzione. Le variabili locali con campo di memoria automatico devono essere sempre esplicitamente inizializzate (altrimenti, il loro valore iniziale è indefinito.)

Variabili definite in una funzione (1/3)

- Le variabili definite in una funzione sono o i parametri formali nella definizione della funzione o le variabili definite nel corpo della funzione (dette anche **variabili locali**).
- Le variabili corrispondenti ai parametri formali di una funzione hanno un **campo di memoria automatico**. Ciò significa che tali variabili vengono riallocate (create) ogni volta che una funzione viene eseguita (tramite chiamata di funzione), esistono finchè l'esecuzione della funzione è attiva e sono distrutte quando la funzione restituisce il controllo al chiamante.
- Per **default**, le variabili locali sono gestite come i parametri formali e, cioè, hanno un campo di memoria automatico. Dunque, i loro valori non sono preservati tra chiamate consecutive alla funzione. Le variabili locali con campo di memoria automatico devono essere sempre esplicitamente inizializzate (altrimenti, il loro valore iniziale è indefinito).

Variabili definite in una funzione (2/3)

- È possibile specificare un **campo di memoria statico** per una variabile locale premettendo nella definizione della variabile lo *specificatore della classe di memoria static* (**variabili locali statiche**)
- Come nel caso delle variabili locali automatiche, quelle statiche sono locali al blocco di codice in cui sono dichiarate (e dunque sono accessibili solo all'interno di esso). La differenza consiste nel fatto che le variabili statiche hanno durata estesa a tutto il tempo di esecuzione del programma. Come le variabili globali, esse sono allocate all'inizio dell'esecuzione del programma e vengono inizializzate solo una volta.
- Ne segue che i valori in esse contenuti sono persistenti: se la funzione viene richiamata, hanno il valore posseduto al termine dell'esecuzione della chiamata precedente.

Variabili definite in una funzione (3/3)

```
void Incrementa()
{
    int x=0;
    x++;
    printf("%d\n",x);
}

int main()
{
    Incrementa();
    Incrementa();
    Incrementa();|
```

Produce come output:
1
1
1

```
void Incrementa()
{
    static int x=0;
    x++;
    printf("%d\n",x);
}

int main()
{
    Incrementa();
    Incrementa();
    Incrementa();|
```

Produce come output:
1
2
3

Spazio dei nomi per variabili dichiarate in funzioni(1/3)

- Le variabili locali (**automatiche o statiche**), i parametri formali di una funzione, e le dichiarazioni extern di variabili all'interno di una funzione hanno **campo di azione o scope esteso al blocco**. Per blocco si intende un qualunque insieme di istruzioni delimitate da {} (corpo di una funzione, istruzioni composte, ecc.). La visibilità inizia dal punto del blocco più interno in cui è dichiarata la variabile fino alla fine del blocco, fine indicata da }.
- Per i parametri formali (anch'essi considerati variabili locali della funzione), il blocco si riferisce all'intero corpo della funzione.
- Dunque, variabili locali aventi lo stesso nome ma dichiarate in funzioni diverse non sono correlate. La stessa cosa vale per i parametri formali delle funzioni.
- Qualsiasi blocco di istruzioni può contenere definizioni di variabili (variabili locali automatiche o statiche).

Spazio dei nomi per variabili dichiarate in funzioni(2/3)

- Quando i blocchi sono annidati e un identificatore definito in un blocco esterno ha lo stesso nome di un identificatore definito in un blocco interno, l'identificatore nel blocco esterno è nascosto fino alla fine del blocco interno. Ciò significa che durante l'esecuzione del blocco interno, il blocco interno vede il valore del proprio identificatore locale e non il valore dell'identificatore dal nome identico e definito nel blocco più esterno.
- Stesso discorso si applica ad identificatori di variabili locali che hanno lo stesso nome di identificatori di variabili globali e le cui dichiarazioni sono visibili al blocco associato alla variabile locale.
- Gli identificatori nella lista dei parametri del prototipo di una funzione hanno **campo d'azione esteso al prototipo di funzione**. Come precedentemente accennato, i prototipi di funzione **non** richiedono nomi nella lista dei parametri: sono necessari solo i tipi. Se si usa un nome nella lista dei parametri del prototipo di funzione, il compilatore *ignora* il nome.

Spazio dei nomi per variabili dichiarate in funzioni(3/3)

```
int x= 1; // variabile globale

void Prova()
{
    int x=0; //Variabile locale automatica

    for(int i=1;i<=2;i++)
    {
        static int x=1; //variabile locale statica
        x++;
        printf("variabili statica %d\n",x);
    }
    printf("variabili automatica %d\n",x);

}

int main()
{
    printf("variabili globale %d\n",x);
    Prova();
}
```

Produce come output:

```
variabili globale 1
variabili statica 2
variabili statica 3
variabili automatica 0
```

Gestione dello stack di chiamata/ritorno di funzioni (1/3)

- Il linguaggio C utilizza una **pila** o **stack**, chiamata anche **pila o stack di esecuzione del programma**, come struttura dati per implementare il meccanismo di chiamata e ritorno dalle funzioni e la creazione, il mantenimento e la distruzione della variabili locali automatiche e parametri formali di tutte le funzioni chiamate.
- Come una pila di piatti, uno stack ha una cima (top dello stack) e un fondo (bottom dello stack). Sono supportate due operazioni:
l'operazione di push che inserisce un nuovo elemento sulla cima e
l'operazione di pop che rimuove l'elemento che si trova sulla cima dello stack.
- Lo stack è una struttura dati che supporta una politica di gestione **last-in first out (LIFO)**: l'*ultimo* elemento inserito nello stack è il *primo* elemento ad essere rimosso da esso.

Gestione dello stack di chiamata/ritorno di funzioni (2/3)

- Quando una funzione è chiamata, essa può chiamare altre funzioni, le quali possono a loro volta chiamarne altre; tutto *prima* che una funzione ritorni restituendo il controllo alla funzione chiamante. Così è necessario tenere traccia degli **indirizzi di ritorno** che servono a ogni funzione per tornare alla funzione che l'ha chiamata.
- La gestione delle chiamate e dei ritorni segue una politica LIFO dal momento che l'*ultima* funzione attivata è anche la *prima* a restituire il controllo.
- Pertanto ogni volta che una funzione chiama un'altra funzione, con un operazione di push, un nuovo elemento (chiamato **record di attivazione**) viene inserito in cima allo stack di esecuzione del programma. Tale record di attivazione è un'area di memoria contenente l'**indirizzo di ritorno** che serve alla funzione chiamata per tornare alla funzione chiamata e le copie delle variabili locali automatiche e i parametri formali associati alla specifica attivazione di funzione.

Gestione dello stack di chiamata/ritorno di funzioni (3/3)

- Quando quella funzione torna alla funzione chiamante, con un'operazione di pop l'associato record di attivazione è rimosso dalla pila e le relative variabili locali automatiche non esistono più per il programma.
- La quantità di memoria in un computer è finita, così si può usare solo una certa quantità di memoria per memorizzare i record di attivazione nella pila di esecuzione del programma. Se vi sono più chiamate di funzione attive di quanti record di attivazione possono essere memorizzati nella pila di esecuzione del programma (ciò può accadere ad esempio per chiamate di funzioni ricorsive), si verifica a tempo di esecuzione un errore irreversibile chiamato **stack overflow**.

Stack delle chiamate di funzioni in azione (1/4)

Esempio: consideriamo come lo stack delle chiamate supporta l'esecuzione di una funzione *square* chiamata dal *main*.

```
int square(int); // prototipo per la funzione square

int main()
{
    int a = 10; // variabile automatica locale in main

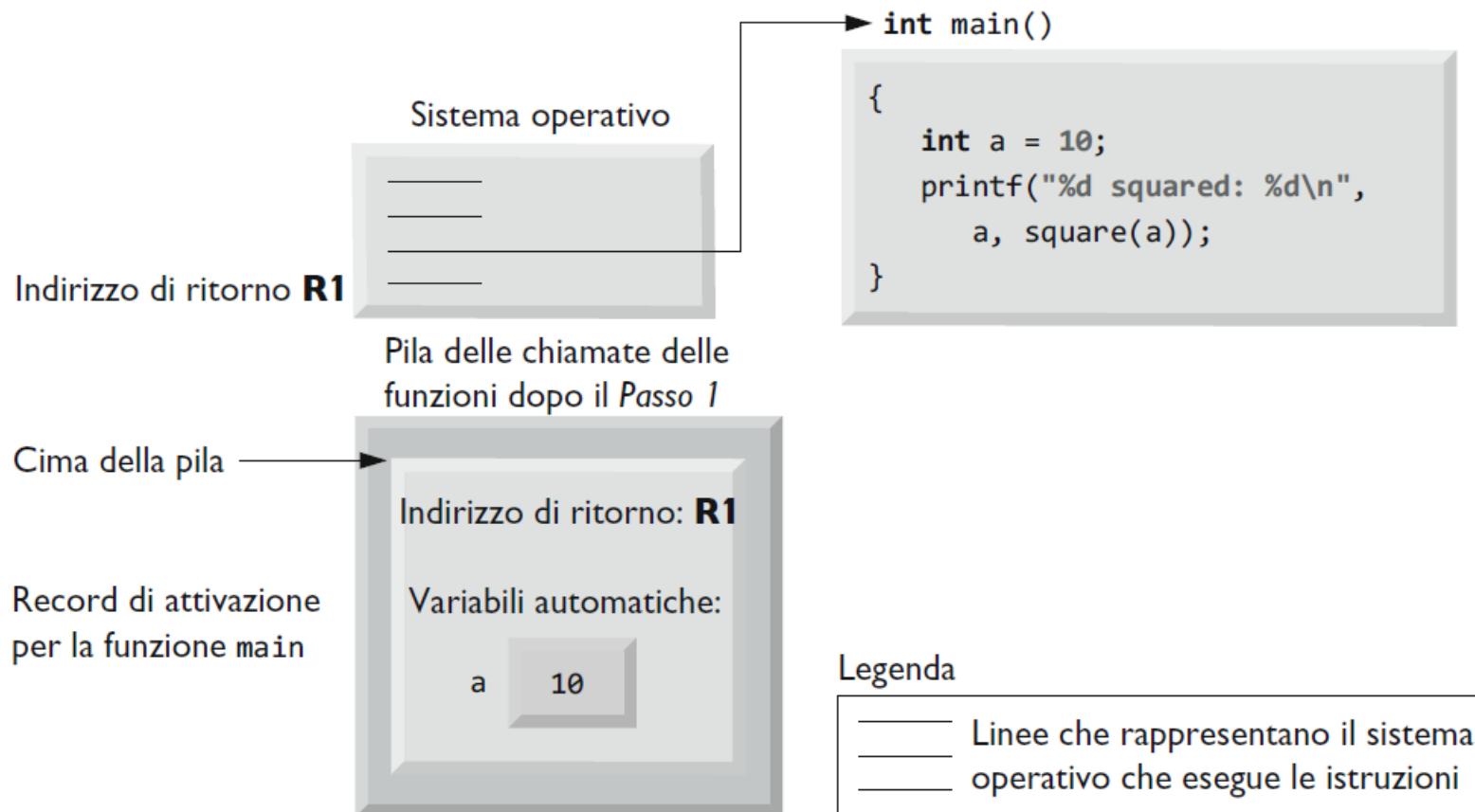
    printf("%d squared: %d\n", a, square(a));
}

// restituisce il quadrato di un intero
int square(int x) // x e' una variabile locale
{
    return x * x; // calcola il quadrato e restituisci il risultato
}
```

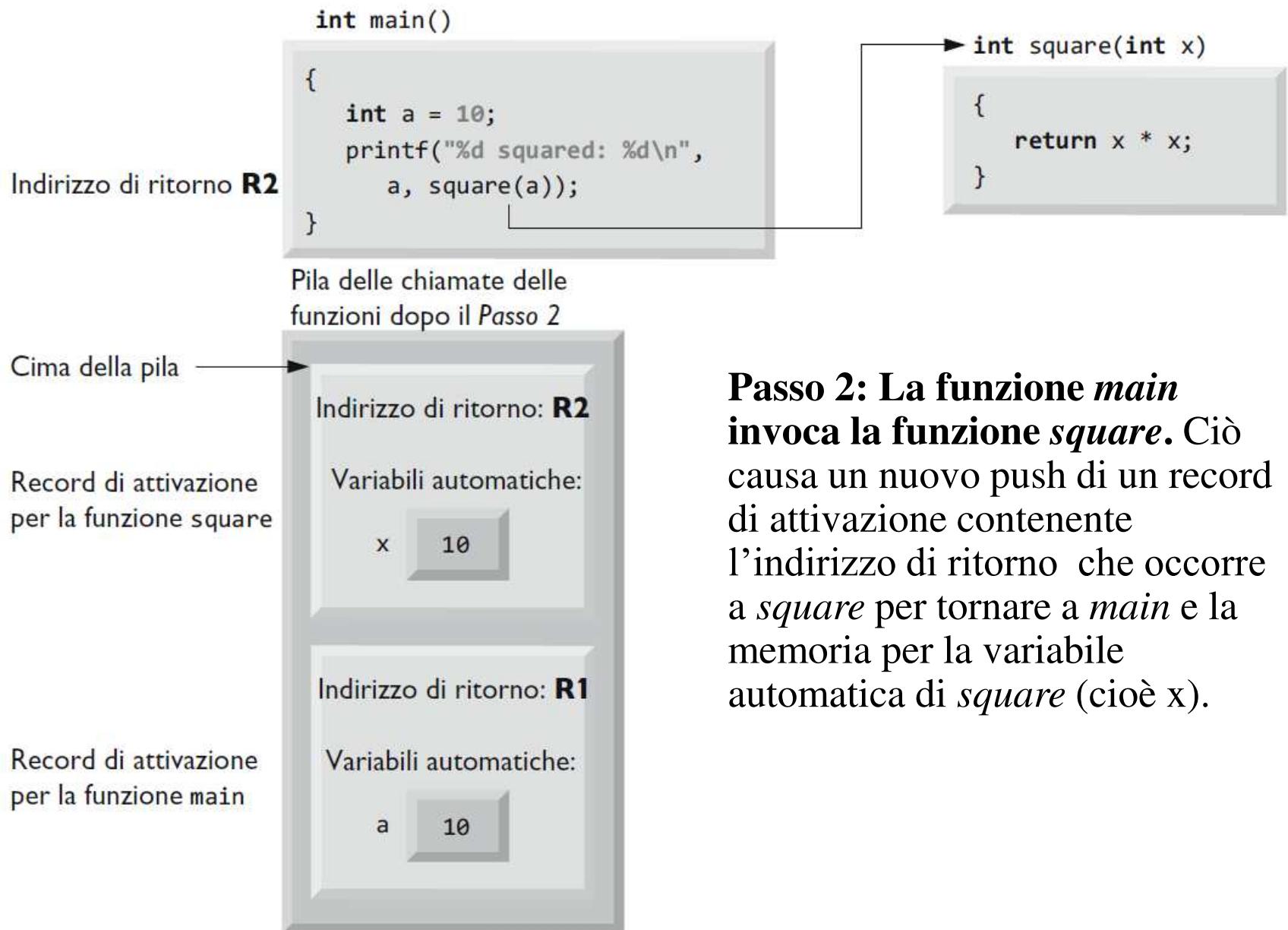
Stack delle chiamate di funzioni in azione (2/4)

Per prima cosa il sistema operativo chiama la funzione *main*. Ciò implica un push di un record di attivazione nella pila che indica alla funzione *main* come tornare al sistema operativo e contiene lo spazio per l'unica variabile automatica di *main* (cioè a, che è inizializzata a 10).

Passo 1: Il sistema operativo invoca *main* per eseguire l'applicazione.

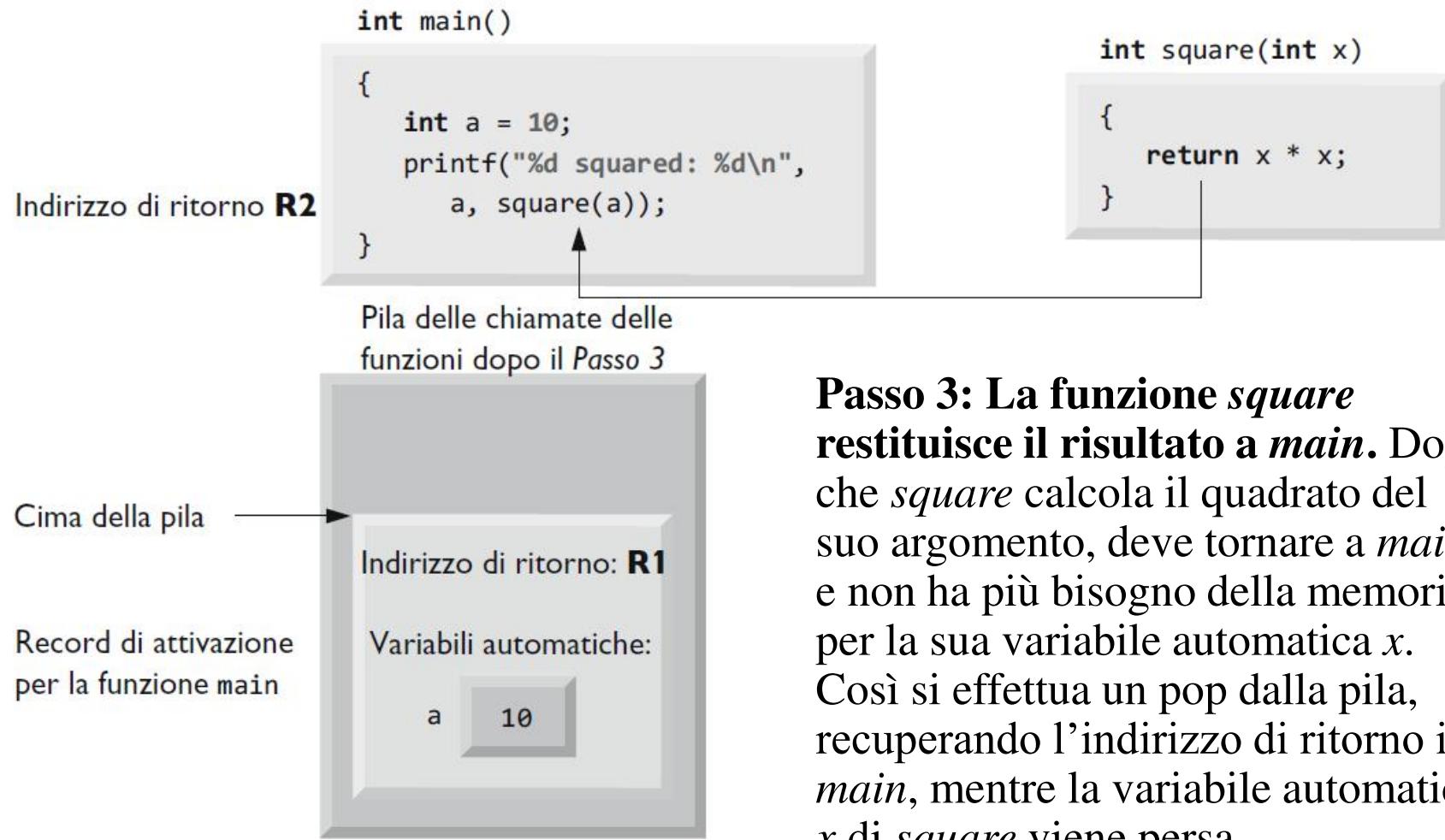


Stack delle chiamate di funzioni in azione (3/4)



Passo 2: La funzione *main* invoca la funzione *square*. Ciò causa un nuovo push di un record di attivazione contenente l'indirizzo di ritorno che occorre a *square* per tornare a *main* e la memoria per la variabile automatica di *square* (cioè x).

Stack delle chiamate di funzioni in azione (4/4)



Passo 3: La funzione `square` restituisce il risultato a `main`. Dopo che `square` calcola il quadrato del suo argomento, deve tornare a `main` e non ha più bisogno della memoria per la sua variabile automatica `x`. Così si effettua un pop dalla pila, recuperando l'indirizzo di ritorno in `main`, mentre la variabile automatica `x` di `square` viene persa.

Risoluzione di problemi computazionali

- Un problema computazionale dipende in generale da un insieme di informazioni (dati) che rappresentano il dominio del problema.
- Un'**istanza del problema** corrisponde al problema stesso per valori specifici dei dati.
- Un algoritmo che risolve uno specifico problema utilizza determinate strutture dati per rappresentare il dominio del problema e determinate operazioni eseguite in un certo ordine per elaborare i dati del dominio.

Risoluzione ricorsiva di problemi (1/3)

Un problema computabile può essere risolto in **modo ricorsivo** se:

- è possibile associare in modo computabile e semplice a ciascuna istanza del problema (valori specifici dei dati del dominio) un numero naturale N (**dimensione dell'istanza**). Ad esempio, per un problema di ordinamento di una sequenza di nomi, la dimensione può essere la lunghezza della sequenza.
- Si dispone di un algoritmo in grado di risolvere in modo diretto il problema per istanze aventi una dimensione più piccola di una certa fissata dimensione N_0 (soglia). Chiameremo tale algoritmo, **algoritmo per i casi base**.
- Si dispone di un algoritmo in grado di computare da un'istanza **I** del problema di dimensione $N > N_0$ un certo numero di istanze **J** dello stesso problema e di dimensione strettamente inferiore a N e di computare la soluzione dell'istanza **I** in termini delle soluzioni delle istanze più piccole **J**. Chiameremo tale algoritmo, **algoritmo di combinazione**.

Risoluzione ricorsiva di problemi (2/3)

Un **algoritmo ricorsivo AR** per risolvere il problema precedente procede allora come segue:

- Dato un input I (istanza del problema), se la dimensione di I è minore o uguale al valore soglia viene applicato l'**algoritmo per i casi base**.
- Altrimenti, in accordo all'**algoritmo di combinazione**, si computano da I le istanze J del problema di dimensione più piccola, si riapplica ricorsivamente l'**algoritmo AR** a ciascuna di queste istanze e si combinano le soluzioni ottenute in accordo all'algoritmo di combinazione.
- Assumendo che l'algoritmo per i casi base e l'algoritmo di combinazione siano corretti, dal principio di induzione matematica ne consegue la correttezza (e anche la terminazione) dell'**algoritmo AR**.

Risoluzione ricorsiva di problemi (3/3)

Consideriamo il problema di calcolare il fattoriale $k!$ di un numero naturale k (per convenzione il fattoriale di 0 è 1):

- Il fattoriale di k può essere definito induttivamente come segue: se k è 0 o 1, allora $k!$ coincide con 1 (passo base). Altrimenti (passo induttivo), $k! = k * (k-1)!$
- Un algoritmo ricorsivo per il fattoriale utilizza come dimensione il dato k . L'algoritmo per i case base computa la soluzione quando k è 0 o 1. Mentre, l'algoritmo di combinazione, a partire dall'istanza k , considera l'istanza $k-1$ e moltiplica il valore di k per la soluzione (calcolata ricorsivamente) dell'istanza $k-1$.

Funzioni ricorsive

Il linguaggio C supporta **funzioni ricorsive** e, cioè, funzioni che chiamano se stesse direttamente, o indirettamente attraverso altre funzioni. Le funzioni ricorsive sono utilizzate per implementare algoritmi ricorsivi. Come esempio, consideriamo la seguente funzione ricorsiva per il calcolo del fattoriale di un numero naturale k.

```
unsigned int fatt(unsigned int k)
{
    int result=0;
    if(k<=1)
        result = 1; //caso base
    else{
        result = fatt(k-1); /* chiamata ricorsiva sull'istanza del
                             problema di dimensione k-1*/
        result = k * result; /* soluzione dell'istanza di dimensione k
                            sulla base della soluzione per */
    }
    return result;
}
```

Calcolo iterativo del fattoriale

Il fattoriale di un numero naturale k può essere calcolato **iterativamente** (non ricorsivamente) utilizzando un'istruzione di iterazione for.

```
unsigned int fattIterativo(unsigned int k)
{
    int result=1;
    for(int i=1;i<=k;i++)
        result = i* result;

    return result;
}
```

La ricorsione rispetto all'iterazione (1/3)

Abbiamo visto con un semplice esempio, come sia possibile risolvere un problema computabile o tramite un **approccio ricorsivo** o tramite un **approccio iterativo**:

- L'approccio iterativo si basa sull'utilizzo di una struttura di controllo di iterazione, mentre l'approccio ricorsivo si basa sull'utilizzo di funzioni ricorsive e strutture di controllo di selezione (per distinguere i casi base dai casi da risolvere ricorsivamente).
- Sia l'iterazione che la ricorsione implicano la *ripetizione*. L'iterazione usa esplicitamente un'istruzione di iterazione. La ricorsione attua la ripetizione attraverso ripetute chiamate di funzione.
- L'iterazione e la ricorsione richiedono ciascuna un *test di terminazione*. L'iterazione termina quando la condizione di continuazione del ciclo diventa falsa, la ricorsione quando si incontra un caso base.

La ricorsione rispetto all'iterazione (2/3)

- L'iterazione controllata da contatore e la ricorsione procedono ciascuna gradualmente verso la terminazione. L'iterazione continua a modificare un contatore finchè questo assume un valore che fa fallire la condizione di continuazione del ciclo. La ricorsione continua a produrre istanze del problema mano a mano più piccole dell'istanza originaria finchè non viene raggiunto il caso base.
- Sia l'iterazione che la ricorsione possono *andare avanti all'infinito*. Si ha un ciclo infinito con l'iterazione se il test di continuazione del ciclo non diventa mai falso. Si ha una ricorsione infinita se il passo di ricorsione *non* riduce ogni volta la dimensione dell'istanza in modo che si converga al caso base. L'iterazione e la ricorsione infinita solitamente si verificano come risultato di errori nella logica di un programma.

La ricorsione rispetto all'iterazione (3/3)

- La ricorsione ha molti difetti. Essa invoca ripetutamente il meccanismo di chiamata di funzione, con conseguente overhead di calcolo che può essere dispendioso sia in termini di tempo di elaborazione che in termini di spazio di memoria. Ogni chiamata ricorsiva comporta l'allocazione di ulteriore spazio di memoria per memorizzare nuove copie delle variabili locali automatiche. Ad esempio, per il calcolo ricorsivo del fattoriale di $k > 1$, possono essere attive $(k-1)!$ chiamate della funzione con conseguenti $(k-1)!$ copie delle variabili locali. **Allora perché utilizzare la ricorsione tenendo anche conto che qualsiasi problema risolvibile ricorsivamente si può anche risolvere iterativamente???**
- Un approccio ricorsivo si preferisce normalmente a un approccio iterativo quando rispecchia in maniera più naturale il problema e produce un programma più facile da capire e da leggere.
- Un'altra ragione per scegliere una soluzione ricorsiva sta nel fatto che potrebbe essere difficile realizzare una soluzione iterativa.

Lezione 10: Esercizi

Laura Bozzelli
a.a. 2020/2021

Funzioni comuni della Libreria math (1/2)

Funzione	Descrizione	Esempio
<code>sqrt(x)</code>	radice quadrata di x	<code>sqrt(900.0)</code> è uguale a 30.0 <code>sqrt(9.0)</code> è uguale a 3.0
<code>cbrt(x)</code>	radice cubica di x (solo per il C99 e il C11)	<code>cbrt(27.0)</code> è uguale a 3.0 <code>cbrt(-8.0)</code> è uguale a -2.0
<code>exp(x)</code>	funzione esponenziale e^x	<code>exp(1.0)</code> è uguale a 2.718282 <code>exp(2.0)</code> è uguale a 7.389056
<code>log(x)</code>	logaritmo naturale di x (in base e)	<code>log(2.718282)</code> è uguale a 1.0 <code>log(7.389056)</code> è uguale a 2.0
<code>log10(x)</code>	logaritmo di x (in base 10)	<code>log10(1.0)</code> è uguale a 0.0 <code>log10(10.0)</code> è uguale a 1.0 <code>log10(100.0)</code> è uguale a 2.0
<code>fabs(x)</code>	valore assoluto di x come numero in virgola mobile	<code>fabs(13.5)</code> è uguale a 13.5 <code>fabs(0.0)</code> è uguale a 0.0 <code>fabs(-13.5)</code> è uguale a 13.5
<code>ceil(x)</code>	arrotonda x all'intero più piccolo non minore di x	<code>ceil(9.2)</code> è uguale a 10.0 <code>ceil(-9.8)</code> è uguale a -9.0

Funzioni comuni della Libreria math (2/2)

<code>floor(x)</code>	arrotonda x all'intero più grande non maggiore di x	<code>floor(9.2)</code> è uguale a 9.0 <code>floor(-9.8)</code> è uguale a -10.0
<code>pow(x, y)</code>	x elevato alla potenza y (x^y)	<code>pow(2, 7)</code> è uguale a 128.0 <code>pow(9, .5)</code> è uguale a 3.0
<code>fmod(x, y)</code>	resto di x/y come numero in virgola mobile	<code>fmod(13.657, 2.333)</code> è uguale a 1.992
<code>sin(x)</code>	funzione trigonometrica seno di x (x in radianti)	<code>sin(0.0)</code> è uguale a 0.0
<code>cos(x)</code>	funzione trigonometrica coseno di x (x in radianti)	<code>cos(0.0)</code> è uguale a 1.0
<code>tan(x)</code>	funzione trigonometrica tangente di x (x in radianti)	<code>tan(0.0)</code> è uguale a 0.0

Generazione di numeri casuali (1/2)

- Tramite la funzione **rand** della libreria standard del C il cui prototipo è contenuto nel file header <stdlib.h>
- La funzione **rand** non prende dati in input e genera in modo casuale un intero non negativo tra 0 e **RAND_MAX**, costante simbolica definita in stdlib.h che assume un valore intero pari ad almeno 32767 (valore intero massimo per un intero a due byte).
- Ogni numero tra 0 e **RAND_MAX** ha un'uguale probabilità di essere scelto ogni volta che **rand** viene chiamata.
- Per cambiare l'intervallo di valori [A,B] di interi non negativi da selezionare in modo casuale con $A \leq \text{RAND_MAX}$, utilizziamo l'espressione **A + (rand() % (B-A +1))**.

Generazione di numeri casuali (2/2)

- La funzione **rand** genera numeri pseudocasuali. Chiamare ripetutamente **rand** genera una sequenza di numeri che *appaiono* casuali. Ogni volta che viene eseguito il programma viene generata la stessa sequenza.
- Per assicurare che anche la sequenza sia generata in modo casuale (randomizzazione) si utilizza la funzione **srand** che prende in ingresso un **unsigned int** (prototipo nel file <stdlib.h>) e fornisce un **seme** alla funzione **rand** per generare una sequenza di numeri casuali diversi per ogni esecuzione del programma. Chiamare **srand** una sola volta prima delle invocazioni a **rand**.
- Per randomizzare senza inserire ogni volta un seme, si può usare l’istruzione **srand(time(NULL))**, dove la funzione **time** (prototipo in <time.h>) restituisce il numero di secondi trascorsi dalla mezzanotte dell’ 1 Gennaio 1970.

Esercizi

1. Un numero non negativo è **perfetto** se i suoi divisori, compreso 1 (ma non il numero stesso), hanno come somma il numero stesso. Ad esempio, 6 è un numero perfetto perché $6 = 1 + 2 + 3$. Scrivete una funzione che prenda in input un intero negativo e determini se l'intero è un numero perfetto. Usate questa funzione in un programma che determini e stampi tutti i numeri perfetti tra 1 e 1000. Stampate i fattori di ogni numero perfetto per confermare che il numero è effettivamente perfetto. **Curiosità:** ad oggi, si conoscono solo 51 numeri perfetti, il più grande dei quali ha 49 724 095 cifre!.
2. Scrivere una funzione che dati in input due interi restituisce il **massimo comun divisore** tra i due interi.
3. Scrivere una funzione ricorsiva che dati in input un numero reale $b \neq 0$ ed un intero non negativo N calcoli la potenza b^N . Utilizzare il fatto che $b^0 = 0$ e $b^{N+1} = b \times b^N$.

Esercizi

4. Scrivere una funzione che dato un intero positivo in input N, e utilizzando le funzioni della libreria standard per la generazione di numeri casuali, simuli il lancio di un dado per N volte, e stampi a video per ogni possibile uscita (valore da 1 a 6) il numero delle volte in cui quell'uscita si è verificata insieme al valore atteso $N/6$. Chiamare la funzione in un ciclo del main che ad ogni passo chiede all'utente di inserire da tastiera un intero non negativo N, fino a quando l'utente non inserisce un valore sentinella (uscita dal ciclo). Verificare che per grandi valori i valori stampati si avvicinano ai valori attesi $N/6$.
5. La serie di Fibonacci 0, 1, 1, 2, 3, 5, 8, 13, 21, ... inizia con i termini 0 e 1 e ha la proprietà che ogni termine che segue è la somma dei due termini precedenti. Scrivete una funzione fibonacci(n) non ricorsiva che calcoli l' n^{mo} numero di Fibonacci. Usate **unsigned int** per il tipo del parametro della funzione e **unsigned long**. Inoltre, determinate il numero di Fibonacci più grande che può essere stampato sul vostro sistema.

Esercizi

6. Cosa fa il seguente programma?

```
#include <stdio.h>

unsigned int Calcola(unsigned int a, unsigned int b);

int main(void)
{
    printf("Inserisci due interi non negativi: \n");
    unsigned int x; //primo intero
    unsigned int y; //secondo intero
    scanf("%u%u", &x, &y);

    printf("Il risultato e' %u\n", Calcola(x, y));
}

unsigned int Calcola(unsigned int a, unsigned int b)
{
    if (0 == b) {      // caso di base
        return 0;
    }
    else if(1 == b){  // caso di base
        return a;
    }
    else { // passo ricorsivo
        return a + Calcola(a, b - 1);
    }
}
```

Lezione 12: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi

1. Scrivere un programma che chieda ad uno studente il numero di esami fatti e i voti ottenuti in ciascun esame. Dopodiché calcoli e stampi a video la media di tali voti (la media deve essere calcolata come numero reale) e stampi di nuovo il numero di esami fatti ed i voti ottenuti. **Suggerimento:** memorizzare i voti in un array, definire una funzione che calcoli la media, ed una funzione che stampi l'array dei voti.
2. Scrivere un programma che inserisca in un array N caratteri, con N e gli N caratteri ottenuti da input (tastiera). Dopodiché richieda un carattere c e calcoli e stampi quante volte c è presente nell'array. **Suggerimento:** definire una funzione che permetta di memorizzare gli elementi nell'array, ed una funzione che calcoli il numero di volte in cui un dato carattere (parametro d'input) input è presente in un array (parametro d'input).
3. Dato un array di N interi, scrivere una funzione ricorsiva che restituisca il valore massimo tra i valori pari presenti nell'array. Se non ci sono valori pari restituisca zero.

Esercizi

4. Scrivere una funzione che dati in input un array di interi e la lunghezza dell'array, restituisca 1 se l'array è ordinato e 0 altrimenti. Testare la funzione nel main tramite dati inseriti da tastiera. Implementare una versione ricorsiva della funzione precedente che prenda in input come parametro addizionale un indice dell'array ed esamini il sottoarray a partire dall'indice dato fino alla fine dell'array.
5. **Operazione di shift di un vettore.** Scrivere una funzione che dati in input un array di interi e la sua lunghezza esegua uno spostamento (shift) a sinistra di una posizione dell'array: ogni non-ultimo elemento deve assumere il valore originario dell'elemento successivo, mentre l'ultimo elemento deve essere 0. Ad esempio, lo shift a sinistra dell'array 1, 3,2,7 di lunghezza 4 è 3,2,7,0. Scrivere anche una funzione che esegua invece uno shift a destra (il primo elemento sarà zero). Testare le due funzioni nel main con dati ricevuti da tastiera.

Esercizi

6. Scrivere una funzione che dati in input un array di interi e la lunghezza dell'array, stampi a video per ogni numero intero presente nell'array il numero di occorrenze di tale numero nell'array. Ad esempio, nell'array 2,3,4,2 il numero 2 occorre 2 volte, mentre 3 e 4 occorrono 1 sola volta. Testare la funzione nel main tramite dati inseriti da tastiera.
Suggerimento: utilizzare un array di supporto per poter tenere traccia dei numeri nel primo per cui sono già state calcolate (e visualizzate a video) le occorrenze.
7. **Merging (fusione) di vettori ordinati.** Scrivere una funzione che prende in input due array di interi insieme alle loro lunghezze, supposti entrambi ordinati per valori crescenti, crei un array di lunghezza pari alla somma delle lunghezze dei due array di input contenente tutti e solo gli elementi dei due array in input ordinati per valori crescenti, e stampi tale array a video.

Algoritmi di ordinamento

Ordinare i dati (e, cioè, mettere i dati in ordine crescente o decrescente, sulla base di una o più chiavi di ordinamento) è una delle più importanti applicazioni di calcolo. Ad esempio, le compagnie telefoniche ordinano gli elenchi dei clienti per cognome, e per ognuno di essi, per nome di battesimo in modo che sia facile trovare i numeri telefonici.

Qui consideriamo algoritmi di ordinamento ad una sola chiave. Assumeremo per illustrare due algoritmi di ordinamento che i dati siano numeri reali (ogni dato consiste della sola chiave di ordinamento) e considereremo l'ordinamento per valori crescenti.

Ordinamento per selezione

L'ordinamento per selezione è un algoritmo di ordinamento semplice ma inefficiente. La prima iterazione dell'algoritmo seleziona l'elemento più piccolo nell'array e lo scambia con il primo elemento. La seconda iterazione seleziona il secondo elemento più piccolo (che è il più piccolo di quelli restanti) e lo scambia con il secondo elemento, e così via. Formalmente, l'algoritmo che chiamiamo **SelectionSort** opera come segue:

SelectionSort (Input: array A di lunghezza N)

1. Inizializza i a 0.
2. Calcola l'indice i_{MIN} dell'elemento minimo del sottoarray di A da i a $N-1$.
3. Scambia gli elementi di indice i e i_{MIN} .
4. Incrementa i .
5. Se i è minore di $N-1$, torna al passo 2. Altrimenti, termina.

Ordinamento per selezione

Il passo 2 dell'algoritmo precedente richiede un ciclo di **N-i passi**. Il passo 2 viene ripetuto per valori di **i** che vanno 0 a N-1. Ne consegue che l'algoritmo richiede un numero di operazioni elementari pari a $\sum_{i=1}^N i = O(N^2)$ (quadratico nella lunghezza dell'array).

SelectionSortRicorsivo (Input: array A di lunghezza N, indice i dell'array)

1. Se i coincide con N-1, termina //Caso base
2. Calcola l'indice i_{MIN} dell'elemento minimo del sottoarray di A da i a N-1.
3. Scambia gli elementi di indice i e i_{MIN} .
4. Chiama **SelectionSortRicorsivo(A,i+1)**//passo ricorsivo.

Ordinamento per fusione (merging sort) (1/3)

L'ordinamento per fusione è un algoritmo di ordinamento efficiente ma concettualmente più complesso dell'ordinamento per selezione. La formulazione ricorsiva dell'algoritmo è alquanto naturale. Se l'array di input ha lunghezza 1 (caso base), allora l'algoritmo termina subito (l'array è già ordinato). Altrimenti, l'algoritmo suddivide l'array in due sottoarray della stessa lunghezza, ordinando ognuno di essi ricorsivamente. I due sottoarray ordinati vengono combinati (fusi) per produrre un array totalmente ordinato (utilizzando lo stesso algoritmo per risolvere l'Esercizio 7). Con un numero dispari di elementi, l'algoritmo crea i due sottoarray, di cui uno con un elemento in più dell'altro.

Ordinamento per fusione (merging sort) (2/3)

MergingSortRicorsivo (Input: array A, indici i_{LEFT} e i_{RIGHT} del sotto-array)

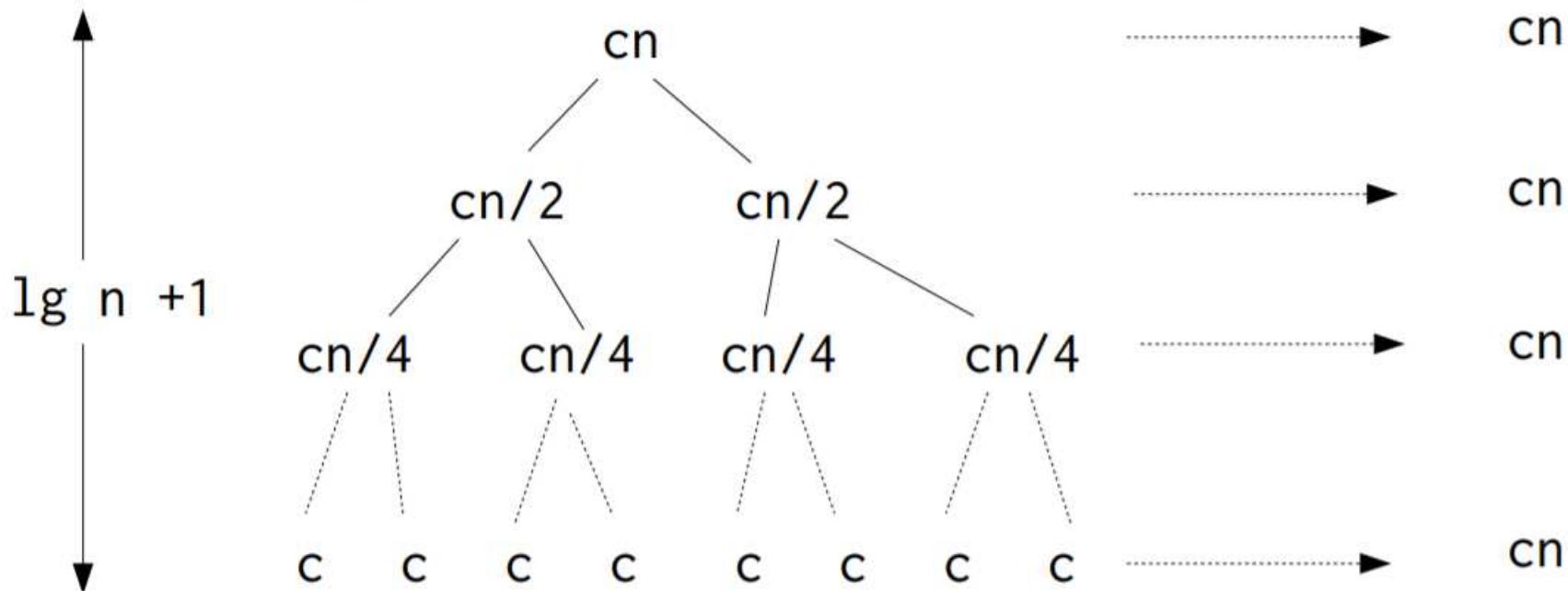
1. Se i_{LEFT} e i_{RIGHT} coincidono termina //Caso base
2. Sia i_{MIDDLE} dato da $i_{LEFT} + (i_{RIGHT} - i_{LEFT})/2$
3. Chiama MergingSortRicorsivo(A, i_{LEFT} , i_{MIDDLE})
4. Chiama MergingSortRicorsivo(A, $i_{MIDDLE}+1$, i_{RIGHT})
5. Applica l'algoritmo di merging per produrre dai sottoarray ordinati $A[i_{LEFT}, i_{MIDDLE}]$ e $A[i_{MIDDLE}+1, i_{RIGHT}]$ un sottoarray complessivo $A[i_{LEFT}, i_{RIGHT}]$ ordinato (implementazione dell'esercizio 7 senza utilizzare un array di supporto).

Inizialmente, per ordinare l'intero array, viene effettuata la chiamata MergingSortRicorsivo(A,0,N-1), dove N è la lunghezza dell'array.

Ordinamento per fusione (merging sort) (3/3)

- Supponendo n sia una potenza di 2 e

$$T(n) = \begin{cases} c & \text{nel caso banale} \\ 2T(n/2) + cn & \text{altrimenti} \end{cases}$$



- Il costo totale è $cn(\lg n + 1) = cn \lg n + cn = \Theta(n \lg n)$

Esercizi

8. Scrivere una funzione che dati in input un array di strutture rappresentanti punti del piano cartesiano e la lunghezza dell'array, ordini l'array per valori crescenti delle ascisse dei punti tramite l'algoritmo SelectionSort e stampi l'array ordinato a video. Testare la funzione nel main tramite dati inseriti da tastiera.
9. Considerare la variante dell'esercizio 8 in cui viene utilizzata la versione ricorsiva del MergeSort basata su un'implementazione dell'esercizio 7 che non utilizzi array di supporto.

Lezione 13

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 13: Puntatori

- Puntatori.
- Operatori di indirizzamento e dereferenziazione.
- Puntatori e funzioni.
- Qualificatore const e suo utilizzo per puntatori.
- Aritmetica dei puntatori.
- Confronto tra puntatori.
- Relazione tra puntatori e array.
- Operatore sizeof
- Operatori bit a bit.

Puntatori (1/3)

- I puntatori rappresentano uno dei costrutti più potenti del linguaggio C.
- Fanno parte delle funzionalità del C più difficili da padroneggiare.
- Consentono di realizzare il passaggio per riferimento favorendo la scrittura di codice compatto ed efficiente.
- Sono fondamentali nel linguaggio C per creare e manipolare **strutture dati dinamiche**, e cioè strutture dati la cui ampiezza può aumentare e diminuire durante l'esecuzione: ad esempio, liste concatenate, code, pile, ed alberi.

Puntatori (2/3)

Abbiamo visto che una variabile è caratterizzata da quattro elementi:

- Nome (identificatore)
- Tipo
- Locazione di memoria
- Valore (contenuto corrente della locazione di memoria)

Un puntatore è una variabile i cui possibili valori sono indirizzi (iniziali) di (locazioni di) memoria.

Puntatori (3/3)

- In generale, un valore ‘valido’ di un puntatore è l’indirizzo di memoria iniziale associato alla locazione di memoria di un’altra variabile. Quest’ultima può essere a sua volta una variabile puntatore (**puntatori di puntatori**), e così via.
- I **puntatori sono tipizzati**: ogni puntatore punta ad un dato di un certo tipo. Fanno eccezione i puntatori a **void** che vengono utilizzati per supportare qualsiasi tipo di puntatore.
- Per ogni tipo di dato predefinito e per ogni tipo aggregato (struttura o unione) o tipo enumerativo, è possibile definire una variabile puntatore di quel tipo.

Dichiarazione di puntatore a dati(1/2)

Sintassi dichiarazione di singola variabile puntatore a dati

<nome tipo> * <nome puntatore>;

- **<nome tipo>**: corrisponde o ad un tipo predefinito, o ad un tipo aggregato (struttura o unione) o ad un tipo enumerativo. **<nome tipo>** può essere anche la parola chiave **void**. I puntatori di tipo void vengono utilizzati per supportare qualsiasi tipo di puntatore.
- **<nome puntatore>**: identificatore valido.

Esempio: dichiarazione di un puntatore ad un intero **int**.

```
int * countPtr;
```

Dichiarazione di puntatore a dati (2/2)

Sintassi dichiarazione di multipli puntatori a dati

<nome tipo> * <nome puntatore 1>, ..., * <nome puntatore N>;

La notazione con l'asterisco * usata per dichiarare puntatori a dati non viene distribuita a tutti i nomi delle variabili in una dichiarazione. Ogni puntatore deve essere dichiarato con il simbolo * che precede il nome.

Esempio:

```
int * xPtr, * yPtr;
```

Dichiarazione di puntatore a puntatore

Sintassi dichiarazione di singolo puntatore a puntatore

<nome tipo> ** <nome puntatore>;

- **<nome tipo>**: corrisponde o ad un tipo predefinito, o ad un tipo aggregato (struttura o unione) o ad un tipo enumerativo. **<nome tipo>** può essere anche la parola chiave **void**.
- **<nome puntatore>**: identificatore valido.

Sintassi dichiarazione di multipli puntatori a puntatore

<nome tipo> ** <nome puntatore 1>, ... , ** <nome puntatore N>;

Esempio:

int ** xPPtr, ** yPPtr;

Inizializzazione e assegnazione di puntatori (1/3)

Un valore ‘valido’ di un puntatore **fa indirettamente riferimento (e, cioè esso punta)** ad un valore avente come tipo il tipo del puntatore e corrispondente al contenuto della locazione di memoria (sequenza di celle di memoria) univocamente determinata da:

- L’indirizzo della prima cella di memoria (informazione fornita dal valore del puntatore).
- Il numero di celle di memoria (informazione fornita dal tipo del puntatore).

Inizializzazione e assegnazione di puntatori (2/3)

I puntatori devono essere inizializzati quando sono dichiarati (alias definiti) oppure assegnando loro un valore tramite un’istruzione di assegnazione. Un valore valido di un puntatore può essere specificato in due modi:

- Tramite l’operatore di indirizzamento & (lo vedremo nelle slide seguenti).
- Tramite chiamate di funzioni di libreria per l’allocazione dinamica di memoria (lo vedremo in seguito).

Inizializzazione e assegnazione di puntatori (3/3)

Ad una variabile puntatore può essere assegnato il valore 0. Il linguaggio C garantisce che **0 non sia mai un indirizzo valido**. Il valore 0 non punta a niente e rappresenta l'unico valore che si può assegnare direttamente a una variabile puntatore.

- La costante simbolica NULL viene spesso usata al posto dello zero come nome mnemonico per indicare che questo, per un puntatore, è un valore speciale.
- La costante simbolica NULL è definita in <stdio.h>.
- Inizializzare un puntatore a 0 equivale ad inizializzare un puntatore a NULL, ma NULL è preferibile poiché evidenzia il fatto che la variabile è un puntatore.

Operatori per i puntatori: & e * (1/5)

Operatore di indirizzamento &

Il C fornisce l'operatore unario **&** per ottenere l'indirizzo di memoria della locazione di memoria associata ad un dato.

- L'operando deve essere un **left value** e, cioè, un'espressione a cui è associata una locazione di memoria e che, dunque, può occorrere come operando sinistro di un'operazione di assegnazione:
 - un nome di variabile,
 - un elemento di un array specificato tramite l'operatore di indicizzazione,
 - il campo di una struttura o di un'unione.
- Il valore restituito dall'operatore **&** può essere assegnato ad una qualsiasi variabile puntatore il cui tipo corrisponde a quello dell'operando a cui è applicato **&**.

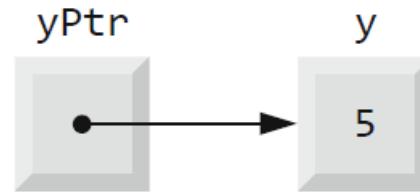
Operatori per i puntatori: & e * (2/5)

Esempio per operatore di indirizzamento &

```
int y=5;  
int * yPtr;
```

//Assegna l'indirizzo della variabile y
// alla variabile puntatore yPtr

```
yPtr = &y;
```



Rappresentazione di y e yPtr in memoria assumendo che l'indirizzo di memoria di y sia 600000 e l'indirizzo di memoria di yPtr sia 500000



Operatori per i puntatori: & e * (3/5)

Operatore di dereferenziazione o di indirezione *

Il C fornisce l'operatore unario * per ottenere il valore della locazione di memoria alla quale punta il suo operando (un puntatore).

- L'operando deve essere una variabile puntatore avente un valore valido (in particolare, diverso da NULL). Applicare l'operatore * ad un puntatore equivale a **deferenziare un puntatore**.

```
int y=5;
int * yPtr;

//Assegna l'indirizzo della variabile y
// alla variabile puntatore yPtr
yPtr = &y;

//Stampa il valore della variabile y, cioè 5
printf("%d", *yPtr);
```

Operatori per i puntatori: & e * (4/5)

Operatore di dereferenziazione o di indirezione *

- Deferenziare un puntatore che non è stato correttamente inizializzato o a cui non è stato assegnato un indirizzo valido (ad esempio, un puntatore il cui valore è NULL) è un errore. Ciò potrebbe determinare un **errore irreversibile** in fase di esecuzione o potrebbe causare la modifica accidentale di dati importanti.
- La deferenziazione di un puntatore può essere gestita alla stessa stregua di una variabile: essa può ricorrere in qualsiasi contesto in cui una variabile del tipo referenziato dal puntatore può occorrere. In particolare, essa può ricorrere come operando sinistro in un'operazione di assegnazione (**left value**). Ad esempio, la deferenziazione di un puntatore di tipo intero può occorrere all'interno di un'espressione numerica.

Operatori per i puntatori: & e * (5/5)

Esempio

```
int x=1, y=2, z[10];
int * ip; //ip è un puntatore ad un intero
ip = &x; //ora ip punta a x
y= *ip //ora y vale 1
*ip = 0; // ora x vale 0
ip =&z[0] // ora ip punta a z[0]
*ip = *ip +10 //incrementa *ip e cioè z[0] di 10
y = *ip +1; // assegna a y il valore di z[0] più 1
*ip += 1; //incrementa di uno l'oggetto puntato da ip e cioè z[0]
(*ip)++; // incrementa di uno l'oggetto puntato da ip e cioè z[0]
```

Poiché i puntatori sono delle variabili essi possono essere usati senza deferimento.

```
//Inizializza il puntatore iq con il
//valore del puntatore ip.
//Dopo l'inizializzazione ip e iq puntano allo stesso oggetto
int * iq = ip;
```

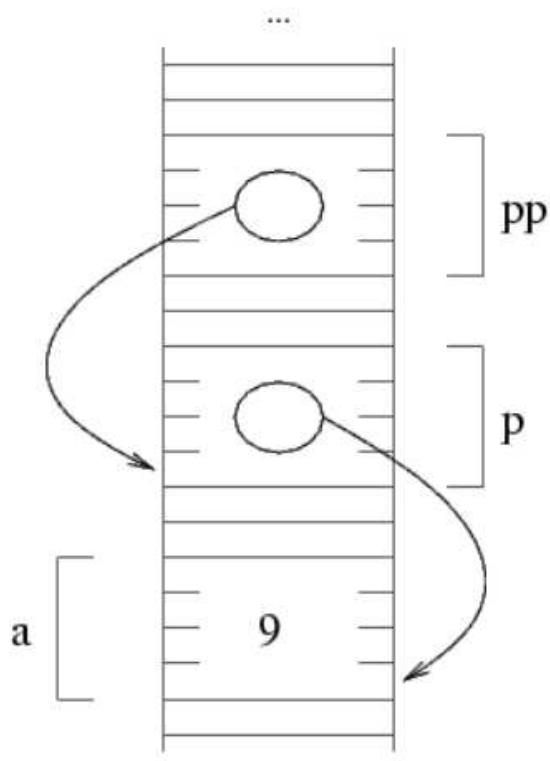
Esempio di puntatore a puntatore (1/2)

Lo specificatore di conversione %p di **printf** invia in uscita l'indirizzo di memoria come un intero esadecimale.

<pre>int main() { int a; int *p; int **pp; a=9; p=&a; //p punta ad a pp=&p; //pp punta a p //Stampa l'indirizzo ed il valore di pp printf("Indirizzo di pp=%p, valore=%p\n", &pp, pp); //Stampa l'indirizzo ed il valore di p printf("Indirizzo di p=%p, valore=%p\n", &p, p); //Stampa l'indirizzo ed il valore di a printf("Indirizzo di a=%p, valore=%i\n", &a, a); return 0; }</pre>	OUTPUT <pre>Indirizzo di pp=000000000062FE08, valore=000000000062FE10 Indirizzo di p=000000000062FE10, valore=000000000062FE1C Indirizzo di a=000000000062FE1C, valore=9</pre>
---	--

Esempio di puntatore a puntatore (2/2)

- In riferimento all'esempio precedente, il valore della variabile **pp** coincide con l'indirizzo della variabile **p**, e il valore di **p** coincide con l'indirizzo di **a**.



- Dato il valore di **pp** è chiaro che è possibile accedere al valore di **a**: basta seguire i puntatori, ossia prima trovare il valore di ***pp**, che è l'indirizzo di **a**, e questo permette di trovare il valore di **a** usando ancora l'operatore *****. Quindi, dato **pp**, il valore di **a** si può trovare con ****pp**.
- In questo modo si può anche assegnare un valore alla variabile **a** usando **pp**: basta usare una istruzione del tipo ****p = ...** .

Deferenziazione di puntatori a dati aggregati (1/2)

I puntatori alle strutture sono usati spesso e, per brevità, è possibile utilizzare una notazione alternativa all'operatore `*` per accedere ai campi di una struttura (o di un'unione) tramite un puntatore alla struttura. Se **p** è un puntatore ad una struttura (o unione) e **campo** è il nome di un membro di una struttura referenziata da **p**, allora l'espressione

p -> campo

consente di accedere al membro **campo** della struttura referenziata da **p** (l'operatore `->` è un segno meno seguito da un maggiore).

p -> campo



(* p).campo

Deferenziazione di puntatori a dati aggregati (2/2)

Esempio:

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};

struct Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

struct Rettangolo r, * rp = &r;
```

Le seguenti espressioni sono equivalenti

```
r.pt1.x
rp->pt1.x
(*rp).pt1.x
```

Puntatori e funzioni (1/8)

- Nel linguaggio C, richiamiamo che gli argomenti nelle chiamate di funzioni **sono passati per valore**: il valore di un argomento formale viene copiato nel corrispondente parametro formale (variabile locale automatica) della funzione.
- Nel linguaggio C, un parametro formale in una definizione di funzione può essere una variabile puntatore. In altri termini, una dichiarazione di un parametro formale può essere una dichiarazione di una variabile puntatore. In questo caso, un corrispondente argomento (parametro attuale) in una chiamata della funzione deve essere un'espressione che restituisce un valore puntatore dello stesso tipo del parametro formale, ad esempio:
 - una variabile dello stesso tipo,
 - o l'applicazione dell'operatore di indirizzamento & ad un'espressione che restituisce un tipo di dato coincidente con il tipo di dato puntato dal parametro formale.

Puntatori e funzioni (2/8)

- In generale, una chiamata consente di modificare il contesto chiamante restituendo un risultato che può essere utilizzato dalla funzione chiamante.
- Quando un parametro formale di una funzione è una variabile puntatore, l'indirizzo di memoria (valore dell'argomento) viene copiato nel parametro formale. In questo modo, tramite il parametro formale, è possibile accedere alla locazione di memoria puntata dall'argomento passato nella chiamata di funzione. In altri termini, è possibile **modificare le variabili nella funzione chiamante** realizzando in modo implicito un **passaggio per riferimento**.
- L'utilizzo di parametri formali di tipo puntatore (implicito passaggio per riferimento) consente ad una funzione di “restituire” più valori alla sua funzione chiamante modificando variabili nella funzione chiamante.

Puntatori e funzioni (3/8)

L'utilizzo di variabili puntatore come parametri formali di funzioni è importante anche per questioni di efficienza:

- Quando un oggetto (ad esempio una struttura) contiene grandi quantità di dati, passare l'oggetto per valore comporta un consumo di tempo e memoria per fare una copia del valore dell'oggetto e trasferirlo al corrispondente parametro formale.
- Passando un puntatore all'oggetto è un'operazione molto più efficiente, dal momento che bisogna copiare solo un indirizzo di memoria.

Puntatori e funzioni (4/8)

Esempio:

```
//Calcola e restituisce il quadrato di un argomento intero
int QuadratoPerValore(int n)
{
    return n*n;
}

//Aggiorna l'intero puntato da pn al
//quadrato del suo valore all'inizio della chiamata
void QuadratoPerRiferimento(int * pn)
{
    *pn = (*pn) * (*pn);
```

OUTPUT

```
Quadrato di number = 25
number = 5
Dopo chiamata a QuadratoPerRiferimento number = 25
```

```
int main()
{
    int number =5;
    printf("Quadrato di number = %d\n",QuadratoPerValore(number));
    //Il valore di number dopo la chiamata a QuadratoPerValore è ancora 5
    printf("number = %d\n",number);
    QuadratoPerRiferimento(&number);
    //Il valore di number dopo la chiamata a QuadratoPerRiferimento è 25
    printf("Dopo chiamata a QuadratoPerRiferimento number = %d",number);
}
```

Puntatori e funzioni (5/8)

Ricordiamo che una variabile array monodimensionale è una variabile puntatore il cui valore è l'indirizzo di memoria del primo elemento di un array.

- Una dichiarazione di un parametro formale corrispondente ad un array monodimensionale può essere equivalentemente sostituita con una dichiarazione di una variabile puntatore del tipo coincidente con il tipo degli elementi dell'array (per il compilatore le due dichiarazioni sono equivalenti).

Sintassi parametro formale array monodimensionale:

<tipo elemento> <nome parametro> []

La dichiarazione precedente è equivalente a

<tipo elemento> * <nome parametro>

Puntatori e funzioni (6/8)

Esempio:

```
void ModificaArray1(int b[], int size)
{
    int i=0;
    for(;i<size;i++)
        b[i] *= 2;
}

void ModificaArray2(int * b, int size)
{
    int i=0;
    for(;i<size;i++)
        b[i] *= 2; //Utilizzo dell'operatore
                    // di indicizzazione per una variabile
                    //puntatore
}
void main()
{
    int A[5] = {1,2,3,4,5};

    ModificaArray1(A,5);
    for(int i=0;i<5;i++)
        printf("Elemento indice %d di A = %d\n",i,A[i]);

    ModificaArray2(A,5);
    for(int i=0;i<5;i++)
        printf("Elemento indice %d di A = %d\n",i,A[i]);
}
```

OUTPUT

```
Elemento indice 0 di A = 2
Elemento indice 1 di A = 4
Elemento indice 2 di A = 6
Elemento indice 3 di A = 8
Elemento indice 4 di A = 10
Elemento indice 0 di A = 4
Elemento indice 1 di A = 8
Elemento indice 2 di A = 12
Elemento indice 3 di A = 16
Elemento indice 4 di A = 20
```

Puntatori e funzioni (7/8)

Una funzione può restituire come risultato un valore di tipo puntatore. In questo caso, per puntatori a dati, l'intestazione della funzione ha il seguente formato:

<tipo valore ritorno> * <nome funzione>(<elenco parametri>)

Restituire un puntatore ad una variabile locale automatica od il nome di un array dichiarato all'interno del corpo della funzione (senza allocarlo in modo dinamico) è un errore! Al termine dell'esecuzione della funzione la variabile locale automatica viene distrutta e l'indirizzo del puntatore restituito non è più valido in generale!

Puntatori e funzioni (8/8)

Esempio:

```
int * BadFunction1(int b)
{
    return &b; //errore
}

int * BadFunction2()
{
    int A[5] = {1,2,3,4,5};
    return &A; //errore
}
```

Qualificatore **const**

Il qualificatore **const** del linguaggio C viene utilizzato nelle dichiarazioni di variabili e nelle dichiarazioni di parametri formali:

- nel caso di dichiarazione di variabili, per specificare che una variabile non possa essere acceduta in scrittura dopo la sua inizializzazione.
- nel caso di dichiarazione di parametri formali, per specificare che il parametro non possa essere acceduto in scrittura nel corpo della funzione.
- Il compilatore segnala un errore in caso contrario!

Qualificatore **const** per dati costanti (1/2)

Quando si usa il qualificatore **const** in una dichiarazione di variabile, l'inizializzazione nella dichiarazione di variabile è necessaria.

Sintassi dichiarazione variabile non puntatore costante:

const <tipo variabile><nome variabile> = <inizializzazione>;

Sintassi dichiarazione parametro formale costante non puntatore:

const <tipo variabile><nome variabile>

Qualificatore const per dati costanti (2/2)

Esempio:

```
//Il parametro b può essere acceduto solo in lettura
void Prova(const int b)
{
    printf("Valore di b =%d",b);
    b=3;//il compilatore segnala un errore
}

void main()
{
    const int c= 5;
    printf("Valore di c =%d",c);
    c= 4;//il compilatore segnala un errore
}
```

Qualificatore **const** per puntatori

Vi sono tre modi per utilizzare il qualificatore **const** per puntatori a dati.

- Puntatori costanti a dati non costanti
- Puntatori non costanti a dati costanti.
- Puntatori costanti a dati costanti.

Puntatore costante a dati non costanti (1/2)

Un **puntatore costante a dati non costanti** punta sempre alla stessa locazione di memoria e i dati in quella locazione *possono essere modificati* per mezzo del puntatore.

Sintassi dichiarazione variabile puntatore costante a dati non costanti:

`<tipo> * const <nome puntatore> = <inizializzazione>;`

Sintassi dichiarazione parametro puntatore costante a dati non costanti:

`<tipo> * const <nome puntatore>`

Puntatore costante a dati non costanti (2/2)

Esempio:

```
void main()
{
    int x=0;
    int y=0;

    //ptr è un puntatore costante ad un intero che può
    //essere modificato tramite ptr
    //ptr è inizializzato con l'indirizzo della
    //variabile intera x
    int * const ptr = &x;

    *ptr = 7; //consentito: aggiorna il valore di x a 7

    //Assegnazione non consentita al valore di ptr:
    //il compilatore genera un error
    ptr = &y; //errore: ptr è costante
}
```

Puntatore non costante a dati costanti (1/3)

Un **puntatore non costante a dati costanti** *può essere modificato* per puntare ad oggetto qualunque del tipo appropriato, ma i *dati* a cui punta *non possono essere modificati* per mezzo del puntatore.

Sintassi dichiarazione variabile puntatore non costante a dati costanti:

`const <tipo> * <nome puntatore>;`

Sintassi dichiarazione parametro puntatore non costante a dati costanti:

`const <tipo> * <nome puntatore>`

Puntatore non costante a dati costanti (2/3)

Esempio:

```
//xPtr non può essere usato per modificare
//la locazione di memoria a cui punta:
//il corpo della funzione non può accedere
//a tale locazione di memoria
void f(const int * xPtr)
{
    *xPtr =100; // errore *xPtr è costante
}

void main()
{
    int y;
    f(&y); //f tenta una modifica non permessa
}
```

Puntatore non costante a dati costanti (3/3)

Parametri formali rappresentati da **puntatori non costanti a dati costanti** consentono di combinare il vantaggio del passaggio per valore (i valori delle variabili del chiamante non possono essere modificati dalla funzione chiamata) con i vantaggi del passaggio per riferimento (efficienza dal momento che viene copiato solo un indirizzo di memoria).

- Ciò può essere utile quando è necessario accedere solo in lettura a dati aggregati (strutture) molto grandi passati dal chiamante alla funzione chiamata. In questo caso, si specifica come parametro formale della funzione chiamata un puntatore a struttura, dichiarato come puntatore non costante a dati costanti. In questo modo, una struttura viene passata per riferimento e ci si assicura allo stesso tempo che la struttura definita nel chiamante non possa essere modificata dal chiamante.

Puntatore costante a dati costanti (1/2)

Un **puntatore costante a dati costanti** punta sempre alla stessa locazione di memoria e i dati in quella locazione di memoria *non possono essere modificati* per mezzo del puntatore. Questo è ad esempio il modo in cui un array deve essere passato ad una funzione che accede solo in lettura agli elementi dell'array.

Sintassi dichiarazione variabile puntatore costante a dati costanti:

`const <tipo> * const <nome puntatore> = <inizializzazione>;`

Sintassi dichiarazione parametro puntatore costante a dati costanti:

`const <tipo> * const <nome puntatore>`

Puntatore costante a dati costanti (2/2)

Esempio:

```
void main()
{
    int x=0;
    int y=0;

    //ptr è un puntatore costante ad un intero costante. ptr
    //punta sempre alla stessa locazione; l'intero in quella
    //locazione non può essere modificato tramite ptr
    const int * const ptr = &x;

    *ptr = 7; //errore: *ptr è costante
    | ptr = &y; //errore: ptr è costante
}
```

Aritmetica dei puntatori (1/5)

È possibile eseguire operazioni '*pseudo*' aritmetiche sui puntatori che consentono di aggiornare l'indirizzo di un puntatore che punta ad un elemento di un array all'indirizzo di un altro elemento dell'array che precede o segue il primo lungo l'array. Il valore ottenuto è generalmente valido solo se esso si riferisce ad un elemento dell'array a cui appartiene l'elemento puntato inizialmente dal puntatore.

Aritmetica dei puntatori (2/5)

Sia `<Ptr>` una variabile puntatore o un'espressione che possa essere utilizzata alla stessa stregua di una variabile puntatore (ad esempio l'applicazione dell'operatore di indirizzamento ad una variabile ordinaria).

- **Operazione di incremento:** l'espressione `<Ptr>++` o `++<Ptr>` *incrementa* `<Ptr>` in modo tale da farlo puntare all'oggetto che segue l'oggetto dello stesso tipo correntemente puntato da `<Ptr>`. Il valore di `++<Ptr>` dipende dal tipo dell'oggetto puntato da `<Ptr>` : il valore viene incrementato del numero di celle di memoria utilizzate per memorizzare gli oggetti del tipo puntabili da `<Ptr>` . L'operazione è generalmente valida solo se `<Ptr>` punta ad un elemento di un array che non è l'ultimo.
- **Operazione di decremento:** l'espressione `<Ptr>--` o `--<Ptr>` *decrementa* `<Ptr>` in modo tale da farlo puntare all'oggetto che precede l'oggetto dello stesso tipo correntemente puntato da `<Ptr>`. L'operazione è generalmente valida solo se `<Ptr>` punta ad un elemento di un array che non è il primo.

Aritmetica dei puntatori (3/5)

Sia **<Ptr>** una variabile puntatore o un'espressione che possa essere utilizzata alla stessa stregua di un puntatore e **<N>** un'espressione numerica che restituisce un intero non negativo.

- **Aggiungere un intero ad un puntatore:** l'espressione **<Ptr> + <N>** indica l' **<N>** -esimo oggetto del tipo puntato da **<Ptr>** che segue quello attualmente puntato da **<Ptr>**. Il valore viene incrementato del numero di celle di memoria utilizzate per memorizzare gli oggetti del tipo puntabili da **<Ptr>** moltiplicato per **<N>**. L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che è seguito da almeno altri **<N>** elementi.
- **Sottrarre un intero ad un puntatore:** l'espressione **<Ptr> - <N>** indica l' **<N>** -esimo oggetto del tipo puntato da **<Ptr>** che precede quello attualmente puntato da **<Ptr>**. L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che è preceduto da almeno altri **<N>** elementi.

Aritmetica dei puntatori (4/5)

Siano **<Ptr>** e **<Qtr>** espressioni dello stesso tipo utilizzabili alla stessa stregua di variabili puntatore:

- **Sottrarre due puntatori:** l'espressione **<Ptr> – <Qtr>** indica il numero di oggetti compresi tra **<Ptr> – <Qtr>** se **<Ptr>** punta ad un oggetto che segue l'oggetto puntato da **<Qtr>** e l'opposto altrimenti.

Le operazioni pseudo aritmetiche sui puntatori viste precedentemente, ad eccezione della sottrazione tra puntatori, possono essere combinate in modo arbitrario per dar luogo ad espressioni complesse che possono essere utilizzate alla stessa stregua di variabili puntatore.

Aritmetica dei puntatori (5/5)

Esempio.

```
#include <stdio.h>
#define N 10

void Inizializza(int *,int);
void Stampa(int *,int);

int main()
{
    int vett[N];
    Inizializza(vett,N);
    Stampa(vett,N);
    return 0;
}
```

```
void Inizializza(int * v,int n)
{
    int i=0;
    for (;i<n;++i)
    {
        printf("Inserisci l'elemento %d:",i+1);
        scanf("%d",&v[i]);
        /*Istruzione alternativa usando
         l'aritmetica dei puntatori:*/
        /* scanf("%d", v+i); */
    }
}

void Stampa(int * v,int n)
{
    int i=0;
    for (i=0;i<n;++i)
        printf("Elemento in posizione %d :"
               " %d\n",i+1,v[i]);
    /*Istruzione alternativa usando
     l'algebra dei puntatori:*/
    /*printf("Elemento in posizione %d :
     %d\n",i+1,*(&v[i])); */
}
```

Confronto fra puntatori

- È possibile confrontare puntatori dello stesso tipo tramite gli operatori relazionali `==`, `!=`, `<`, `<=`, ecc. a patto che i due operatori puntino ad elementi dello stesso array. Il confronto può essere utile per determinare ad esempio se uno dei due punta ad un elemento con indice più alto di quello puntato dall'altro.
- Qualsiasi puntatore può essere confrontato con lo 0 (alias la costante `NULL`) tramite gli operatori `==` e `!=`.
- Gli operatori di confronto possono essere applicati anche ad espressioni puntatore utilizzabili alla stessa stregua di variabili puntatore.

Assegnare puntatori ad altri puntatori

- Un puntatore può essere assegnato ad un altro puntatore se entrambi hanno lo stesso tipo. L'eccezione a questa regola è costituita dal **puntatore a void** (cioè **void ***), il quale è un puntatore generico che può rappresentare qualsiasi tipo di puntatore.
- Ad ogni tipo di puntatore è possibile assegnare un puntatore a **void**, e a un puntatore a **void** è possibile assegnare un puntatore di qualsiasi tipo. In entrambi i casi non è necessaria alcuna operazione di cast.
- Un puntatore a **void** non può essere deferenziato: questo perché un puntatore a **void** punta ad una locazione di memoria per un tipo di dati *sconosciuto*. Il compilatore deve conoscere il tipo di dati per determinare il numero di byte che rappresentano l'oggetto puntato.

Relazione tra puntatori e array (1/3)

Array e puntatori sono intimamente correlati in C e spesso possono essere usati in maniera intercambiabili.

- Per definizione, il valore di una variabile array o di un'espressione array è l'indirizzo del primo elemento dell'array (elemento di indice 0). In effetti, una variabile array si comporta allo stesso modo di un **puntatore costante**. Essa punta sempre all'inizio dell'array e non può essere utilizzata come operando sinistro di un'operazione di assegnazione.
- I puntatori possono essere usati per fare qualsiasi operazione che implichi l'indicizzazione di un array.

Per illustrare ciò, consideriamo le seguenti definizioni:

```
int b[5]; //Array di 5 interi
int * bPtr; //Puntatore ad un intero
```

Relazione tra puntatori e array (2/3)

```
int b[5]; //Array di 5 interi  
int * bPtr; //Puntatore ad un intero
```

Poiché la variabile **b** (nome dell'array) punta al primo elemento dell'array, l'assegnamento:

```
bPtr = &b[0]; //bPtr punta al primo elemento dell'array b
```

può anche essere scritto nella forma.

```
bPtr = b;
```

Inoltre, ad un elemento dell'array, ad esempio **b[3]**, si può fare alternativamente riferimento con l'espressione con puntatori:

```
*(bPtr + 3) //notazione puntatore/offset
```

In generale, l'operazione di indicizzazione di array può essere equivalentemente sostituita dalla notazione puntatore/offset

Relazione tra puntatori e array (3/3)

```
int b[5]; //Array di 5 interi  
int * bPtr; //Puntatore ad un intero
```

I puntatori possono essere indicizzati come le variabili array. Se **bPtr** ha il valore di **b**, allora l'espressione

bPtr[1]

si riferisce all'elemento **b[1]** dell'array.

Riassumendo, possiamo dire che un'espressione sotto forma di array e indici è equivalente ad una che utilizza puntatori e offset.

Riepilogo utilizzo puntatori

Le operazioni consentite sui puntatori sono:

- Dereferenziazione e (ulteriore) indirizzamento.
- Assegnazione fra puntatori dello stesso tipo.
- Addizione/sottrazione fra puntatori ed interi.
- Sottrazione e confronto fra due puntatori ad elementi di uno stesso array.
- Assegnazione e confronto con lo zero (costante NULL).
- Indicizzazione di puntatori.

Operatori sizeof (1/2)

- L'operatore unario predefinito **sizeof** viene utilizzato per determinare l'ampiezza in byte della locazione di memoria associata ad un tipo di dato.
- Deve essere utilizzato con la notazione per chiamate di funzioni. Esempio: **sizeof(int)**
- Può essere applicato al nome di una qualsiasi variabile, all'identificatore di un qualsiasi tipo di dati, e anche ad una generica espressione.
- Quando applicato al nome di un array, restituisce l'ampiezza complessiva dell'array (il numero di elementi per l'ampiezza in byte del tipo di dato associato a ciascun elemento).

Operatori sizeof (2/2)

```
void main()
{
    unsigned int x;
    scanf("%u",&x);

    int A[x];
    printf("Ampiezza dell'array = %u",sizeof(A));
}
```

OUTPUT

```
9
Ampiezza dell'array = 36
```

Operatori bit a bit (1/6)

- Gli operatori bit a bit si applicano a operandi di tipo intero con segno o senza segno.
- Sono utilizzati per manipolare i bit nella rappresentazione binaria di operandi interi. Il primo bit (quello più a sinistra) è il bit più significativo.
- Il linguaggio C fornisce 6 operatori bit a bit:
 - AND bit a bit: simbolo &
 - OR inclusivo bit a bit: simbolo |
 - OR esclusivo bit a bit: simbolo ^
 - Shift (spostamento) a sinistra: simbolo <<
 - Shift (spostamento) a destra: simbolo >>
 - Complemento a uno: simbolo ~

Operatori bit a bit (2/6)

AND bit a bit &

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se entrambi i corrispondenti bit dei due operandi sono 1, e a 0 altrimenti.

Esempio:

Il risultato di combinare

$65535 = 00000000 \ 00000000 \ 11111111 \ 11111111$

$1 = 00000000 \ 00000000 \ 00000000 \ 00000001$

utilizzando l'operatore & è

$1 = 00000000 \ 00000000 \ 00000000 \ 00000001$

Operatori bit a bit (3/6)

OR inclusivo bit a bit |

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se almeno uno dei corrispondenti bit nei due operandi è 1, e a 0 altrimenti.

Esempio:

Il risultato di combinare

$15 = 00000000 \ 00000000 \ 00000000 \ 00001111$

$241 = 00000000 \ 00000000 \ 00000000 \ 11110001$

utilizzando l'operatore | è

$255 = 00000000 \ 00000000 \ 00000000 \ 11111111$

Operatori bit a bit (3/6)

OR esclusivo bit a bit ^

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se i bit corrispondenti nei due operandi sono differenti, e a 0 altrimenti.

Esempio:

Il risultato di combinare

$139 = 00000000 \ 00000000 \ 00000000 \ 10001011$

$199 = 00000000 \ 00000000 \ 00000000 \ 11000111$

utilizzando l'operatore $^$ è

$76 = 00000000 \ 00000000 \ 00000000 \ 01001100$

Operatori bit a bit (4/6)

Complemento a 1 ~

- Operatore unario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se il bit corrispondente dell'operando è 0, e a 0 altrimenti.

Esempio:

Il complemento a 1 di

$21845 = 00000000\ 00000000\ 01010101\ 01010101$

è

$4294945450 = 11111111\ 11111111\ 10101010\ 10101010$

Operatori bit a bit (5/6)

Shift a sinistra <<

- Operatore binario. Sposta a sinistra i bit del suo operando sinistro del numero di bit **N** specificato nel suo operando destro.
- Gli ultimi **N** bit sono sostituiti con 0.
- I primi **N** bit dell'operando vanno perduti.
- Il risultato dello shift a sinistra è indefinito se l'operando destro è negativo o se l'operando destro è più grande del numero di bit con cui è memorizzato l'operando sinistro.

Esempio:

Il risultato dello shift a sinistra << di

$960 = 00000000\ 00000000\ 00000011\ 11000000$

di 8 posizioni bit (e, cioè, $960 \ll 8$) è

$245760 = 00000000\ 00000011\ 11000000\ 00000000$

Operatori bit a bit (6/6)

Shift a sinistra >>

- Operatore binario. Sposta a destra i bit del suo operando sinistro del numero di bit **N** specificato nel suo operando destro.
- I primi **N** bit sono sostituiti con 0.
- Gli ultimi **N** bit dell'operando vanno perduti.
- Il risultato dello shift a destra è indefinito se l'operando destro è negativo o se l'operando destro è più grande del numero di bit con cui è memorizzato l'operando sinistro.

Esempio:

Il risultato dello shift a destra >> di

$960 = 00000000\ 00000000\ 00000011\ 11000000$

di 8 posizioni bit (e, cioè, $960 >> 8$) è

$3 = 00000000\ 00000000\ 00000000\ 00000011$

Esempio utilizzo operatori bit a bit

Stampa di un intero nella sua rappresentazione in bit

```
void VisualizzaBits(unsigned int value)
{
    unsigned int numBits = 8 * sizeof(unsigned int);

    //Alla variabile maschera è assegnato il valore il cui unico bit
    // pari a 1 è il primo
    unsigned int maschera = 1<< (numBits-1);

    //effettua un'iterazione sul numero di bit
    for(unsigned int c=1; c<= numBits;c++)
    {
        //L'operatore & è usato con la maschera per nascondere
        // tutti i bit di value ad eccezione del primo
        //Primo bit del valore corrente di value
        unsigned int bit = (value & maschera) ? 1: 0;
        printf("%d", bit);

        //shift di value a sinistra di un bit
        value <<= 1;

        if(c % 8 ==0)
            printf("  ");
    }
}

void main()
{
    printf("65000 = ");
    VisualizzaBits(65000);
}
```

65000 =	00000000	00000000	11111101	11101000
---------	----------	----------	----------	----------

Operatori di assegnazione bit a bit

Ogni operatore bit a bit binario ha un operatore di assegnazione corrispondente. Questi operatori di assegnazione bit a bit sono usati in modo simile agli operatori di assegnazione aritmetici.

Operatori di assegnazione bit a bit

- &= Operatore di assegnazione AND bit a bit.
- | = Operatore di assegnazione OR inclusivo bit a bit.
- ^= Operatore di assegnazione OR esclusivo bit a bit.
- <<= Operatore di assegnazione di spostamento a sinistra.
- >>= Operatore di assegnazione di spostamento a destra.

Precedenza e associatività degli operatori

Precedenza e associatività dei vari operatori nel linguaggio C:
mostrati dall'alto in basso in ordine decrescente di precedenza.

Operatori	Associatività	Tipo
() [] . -> ++ (postfisso) -- (postfisso)	da sinistra a destra	con precedenza più alta
+ - ++ -- ! & * ~ sizeof (tipo)	da destra a sinistra	unario
* / %	da sinistra a destra	moltiplicativo
+ -	da sinistra a destra	additivo
<< >>	da sinistra a destra	di spostamento
< <= > >=	da sinistra a destra	relazionale
== !=	da sinistra a destra	di uguaglianza
&	da sinistra a destra	AND bit a bit
^	da sinistra a destra	XOR bit a bit
	da sinistra a destra	OR bit a bit
&&	da sinistra a destra	AND logico
	da sinistra a destra	OR logico
? :	da destra a sinistra	condizionale
= += -= *= /= &= = ^= <<= >>= %=	da destra a sinistra	di assegnazione
,	da sinistra a destra	virgola

Lezione 14: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi

1. **Strutture e Array.** Definire un tipo di dato struct di nome **Studente** che abbia 4 campi corrispondenti al cognome, al voto più basso, al voto più alto e alla media dei voti di uno studente. Scrivere un programma che una volta riempiti con i dati necessari N strutture di tipo **Studente** (con N dato da tastiera) (le N strutture vanno memorizzate in un array di lunghezza N) consenta le seguenti scelte:
 - Visualizzare i dati di tutti gli studenti.
 - Cercare uno studente per cognome e visualizzane il voto più basso, quello più alto e la media.
 - Visualizzare i dati dello studente con la media più alta.
 - Uscire dal programma.

Esercizi

2. Nel seguente frammento di codice c'è qualcosa da non fare assolutamente. Dire cosa è e motivarlo opportunamente.

```
//Quale è il potenziale errore a tempo di esecuzione?  
int * InizializzaArray(unsigned int N)  
{  
    if(N==0)  
        return NULL;  
  
    int A[N];  
    int i;  
    for (i=0;i<N;++i)  
    {  
        printf("Inserisci l'elemento A[%d]:",i);  
        scanf("%d",&A[i]);  
    }  
    return A;  
}
```

Esercizi

3. Trovate gli errori nel seguente programma e motivateli opportunamente.

```
//Trova gli errori!
int main()
{
    int *zPtr;
    int *aPtr = NULL;
    void *sPtr = NULL;
    int number;
    int z[5] = {1, 2, 3, 4, 5};
    sPtr = z;
    ++zPtr;
    number = zPtr;
    number = *zPtr[2];
    int i;
    for (i = 0; i <= 5; ++i) {
        printf("%d ", zPtr[i]);
    }
    number = *sPtr;
    ++z;
}
```

Esercizi

4. Scrivere un programma che prenda in input un intero $N > 0$, inizializzi un array intero di lunghezza N con interi inseriti da tastiera e ristampi l'array utilizzando puntatori e l'operatore di dereferenziazione * invece di accedere direttamente agli elementi dell'array tramite l'operatore di indicizzazione.
5. Scrivere una funzione che prenda in input un array di interi e la sua lunghezza e ritorni un puntatore verso l'elemento massimale dell'array.
6. Scrivere una funzione che prenda in input un array di interi e restituisca un puntatore al primo elemento dell'array dopo aver invertito (utilizzando un array di appoggio) la sequenza degli elementi in esso contenuti (ad esempio, l'inverso della sequenza 1,2,3,5 è 5,3,2,1). Si testi la funzione stampando l'array dopo l'inversione ed utilizzando il programma per risolvere l'esercizio 4.

Ricerca in un array

Assumiamo di operare su array di interi. Un problema di ricerca su un array consiste nel determinare se un array contenga un certo valore (**chiave di ricerca**). Per array non ordinati, un algoritmo di ricerca dovrà confrontare **ogni elemento** dell'array con la chiave di ricerca, eseguendo pertanto un numero di operazioni elementari in generale lineare nella lunghezza del dato array (**ricerca lineare**).

Ricerca binaria in un array ordinato (1/2)

Se invece l'array dato è ordinato (assumiamo per valori crescenti), si può usare la tecnica molto veloce di **ricerca binaria**. La formulazione ricorsiva dell'algoritmo di ricerca binaria è alquanto naturale.

- Se l'array di input ha lunghezza 1 (caso base), allora l'algoritmo termina con successo se e solo se l'unico elemento dell'array coincide con la chiave di ricerca.
- Altrimenti, l'algoritmo localizza l'**elemento centrale** dell'array e suddivide l'array in due sottoarray di uguale lunghezza (se la lunghezza dell'intero array è pari), il primo che va dal primo elemento all'elemento centrale e il secondo corrispondente alla rimanente porzione dell'intero array. Allora, la ricerca continua ricorsivamente nel primo sotto-array se la chiave di ricerca è minore o uguale all'elemento centrale, e nel secondo sotto-array altrimenti.

Ad ogni passo ricorsivo, una metà degli elementi del sotto-array correntemente esaminato viene rimossa dalla ricerca. Pertanto, **il numero di operazioni elementari è logaritmico nella lunghezza dell'array**.

Ricerca binaria in un array ordinato (2/2)

RicercaBinariaRicorsiva (Input: array A, indici i_{LEFT} e i_{RIGHT} del sotto-array, chiave di ricerca C)

1. Se i_{LEFT} e i_{RIGHT} coincidono termina e restituisci **SI** se $A[i_{LEFT}]$ coincide con la chiave di ricerca C e **NO** altrimenti //Caso base
2. Sia i_{MIDDLE} dato da $i_{LEFT} + (i_{RIGHT} - i_{LEFT})/2$
3. Se $A[i_{MIDDLE}]$ è maggiore o uguale a C, restituisci RicercaBinariaRicorsiva($A, i_{LEFT}, i_{MIDDLE}, C$); altrimenti, restituisci RicercaBinariaRicorsiva($A, i_{MIDDLE}+1, i_{RIGHT}, C$).

Inizialmente, per effettuare la ricerca binaria sull'intero array, viene effettuata la chiamata RicercaBinariaRicorsiva($A, 0, N-1, C$), dove N è la lunghezza dell'array.

Esercizi

7. Scrivere una funzione ricorsiva che implementi l'algoritmo di ricerca binaria su array ordinati di interi. Testare la funzione nel main tramite dati inseriti da tastiera (eventualmente ordinare l'array tramite uno degli algoritmi di ordinamento già studiati prima di chiamare la funzione ricorsiva).
8. Implementare la variante iterativa dell'esercizio 7.

Lezione 15

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 15: Matrici e stringhe

- Array bidimensionali (matrici).
- Dichiarazione e inizializzazione di matrici.
- Accesso agli elementi di una matrice.
- Matrici e funzioni.
- Matrici verso array di puntatori.
- Stringhe.
- Funzioni di libreria per la manipolazione di caratteri.
- Funzioni di libreria per la manipolazione di stringhe.

Array monodimensionali (1/2)

- Nel linguaggio C, un **array**, detto anche **vettore**, è una sequenza ordinata di dati dello stesso tipo, raggruppati sotto un unico nome.
- I singoli dati in un array monodimensionale sono chiamati **elementi dell'array** e possono essere acceduti tramite l'**operatore di indicizzazione** `[]`.
- Elementi consecutivi in un array hanno **locazioni di memoria contigue**.
- Il numero di elementi di un array monodimensionale è chiamato **lunghezza dell'array**. Gli array sono in generale dati **a lunghezza variabile** dal momento che la lunghezza di un array può essere specificata tramite espressioni numeriche intere il cui valore è noto a tempo di esecuzione.

Array monodimensionali (2/2)

- A differenza degli aggregati (strutture e unioni) e dei tipi enumerativi, per definire una variabile di tipo array non bisogna prima definire un tipo array.
- Nella dichiarazione di una variabile di tipo array monodimensionale si specifica il nome della variabile, il tipo comune degli elementi di un array, e la lunghezza dell'array.
- Il tipo di un elemento di un array monodimensionale può essere un tipo predefinito semplice, un tipo **struct**, un tipo **union**, o un tipo enumerativo.
- Una variabile array è una **variabile puntatore costante**: il suo valore rappresenta **sempre** l'indirizzo di memoria della locazione di memoria associata al primo elemento dell'array.

Array bidimensionali (1/5)

- Un **array bidimensionale** (detto anche **matrice**) o **array di dimensione 2** è un array monodimensionale i cui elementi sono array monodimensionali dello stesso tipo e aventi la stessa lunghezza L.
- I singoli elementi di tipo array di un array bidimensionale sono chiamati **righe** e hanno la stessa lunghezza. Per elemento di un array bidimensionale (o matrice) intendiamo un elemento di qualche riga della matrice.
- La lunghezza complessiva di una matrice corrisponde al numero complessivo dei suoi elementi e, cioè, il numero di righe moltiplicato per la lunghezza di ciascuna riga.

Array bidimensionali (2/5)

- Un uso comune degli array bidimensionali (matrici) è quello di rappresentare **tabelle** di valori che contengono informazioni disposte in *righe* e *colonne*.
- I singoli elementi di un array bidimensionale vengono acceduti specificando tramite il duplice uso dell'operatore di indicizzazione [] due indici:
 - L'indice di riga che va da 0 al numero di righe -1.
 - L'indice di colonna che va da 0 a L-1, dove L è la lunghezza di ciascuna riga.
- Il **numero di colonne** di una matrice corrisponde alla lunghezza di ciascuna riga. Ad esempio, per una matrice A con N righe e M colonne per accedere all'elemento di indice di riga $0 \leq i \leq N-1$ e indice di colonna $0 \leq j \leq M-1$, si utilizza l'espressione $A[i][j]$.

Array bidimensionali (3/5)

Dal punto di vista della memoria, gli array bidimensionali sono gestiti nel linguaggio C come array monodimensionali. Questo significa che tutti gli elementi della matrice sono disposti in memoria come un'unica sequenza ordinata (per indirizzi di memoria) di celle di memoria contigue:

- Gli elementi della prima riga precedono quelli della seconda riga, e così via. Le righe sono ordinate per valori crescenti dell'indice di riga.
- Elementi all'interno di una stessa riga sono ordinati per valori crescenti dell'indice di colonna.

Un array bidimensionale (matrice) con M righe e N colonne è chiamato anche **array o matrice M per N**.

Array bidimensionali (4/5)

Esempio: matrice a 3 per 4

Gli elementi della riga i con $0 \leq i \leq 2$ hanno tutti il primo indice (indice di riga) uguale a i .

Gli elementi della colonna j con $0 \leq j \leq 3$ hanno tutti il secondo indice (indice di colonna) uguale a j .

	Colonna 0	Colonna 1	Colonna 2	Colonna 3
Riga 0	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
Riga 1	$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
Riga 2	$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

Diagramma di indicizzazione:

- Indice di colonna (Indice 0, 1, 2, 3)
- Indice di riga (Indice 0, 1, 2)
- Nome dell'array (a)

Le tre frecce puntano al valore $a[2][1]$, che è l'elemento della seconda riga e della seconda colonna.

Array bidimensionali (5/5)

- Similmente agli array monodimensionali, per definire una variabile di tipo array bidimensionale non bisogna prima definire un tipo array bidimensionale.
- Nella dichiarazione di una variabile di tipo array bidimensionale si specifica il nome della variabile, il tipo comune degli elementi della matrice, il numero di righe, ed il numero di colonne.
- Il tipo di un elemento di un array bidimensionale può essere un tipo predefinito semplice, un tipo **struct**, un tipo **union**, o un tipo enumerativo.

Dichiarazione array bidimensionali (1/3)

Sintassi matrici in C:

<tipo elemento> <nome matrice> [<N_Righe>][<N_Colonne>];

- **<tipo elemento>**: rappresenta il tipo comune degli elementi. Per elementi semplici è l'insieme di parole chiave per identificare il tipo semplice (int, double, char, long double, ecc). Per i tipi struttura (risp., unione, risp., enumerazione) è la parola chiave **struct** (risp., **union**, risp., **enum**) seguita dal nome del tipo (come specificato nella definizione di tipo).
- **<nome matrice>**: nome della matrice (qualsiasi identificatore valido).
- **<N_Righe>**: numero di righe. Può essere una qualsiasi espressione numerica di tipo intero.
- **<N_Colonne>** : numero di colonne. Può essere una qualsiasi espressione numerica di tipo intero.

Dichiarazione array bidimensionali (2/3)

Esempio dichiarazione matrici in C: matrici di strutture

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};

//Matrice di punti 10x20
//10 righe e 20 colonne
struct Punto GrigliaPunti[10][20];
```

Dichiarazione array bidimensionali (3/3)

Sintassi:

<tipo elemento> <nome matrice> [<N_Righe>][<N_Colonne>];

- Per dichiarazioni di (variabili) matrici globali o locali statiche (e, cioè, il cui campo di memoria è statico), le espressioni **<N_Righe>** e **<N_Colonne>** devono essere espressioni numeriche intere costante i cui valori, determinabili a tempo di compilazione, devono essere interi positivi. Altrimenti, il compilatore segnala un errore di sintassi. Per tali variabili, l'area di memoria viene allocata all'inizio dell'esecuzione del programma.
- Per dichiarazioni di matrici locali automatiche, è possibile utilizzare espressioni numeriche intere generiche per specificare il numero di righe ed il numero di colonne. La valutazione a tempo di esecuzione dell'espressione numerica per il numero di righe e numero di colonne può generare un errore irreversibile se il valore ottenuto non è positivo e eccede la capacità di memoria a disposizione.

Lista di inizializzazione per matrici (1/3)

Dichiarazione matrice con lista di inizializzazione:

<N_Righe> e **<N_Colonne>** devono essere espressioni intere costanti.

Sintassi <inizializzazione>: rappresenta la parte di inizializzazione.

{ <Init 1>,, <Init N> }

- N deve essere non superiore al numero di righe.
 - **<Init i>** inizializza la righe di indice i-1: è una lista di inizializzazione completa o parziale per la riga di indice i che specifica i valori dei singoli elementi separati da virgole e racchiusi tra parentesi graffe
 - Tutti gli elementi che **non** hanno un inizializzatore esplicito sono inizializzati automaticamente a zero.
 - Per **dichiarazioni di variabili globali o statiche**, le espressioni numeriche utilizzate per l'inizializzazione dei singoli elementi devono essere costanti.

List di inizializzazione per matrici (2/3)

Esempio:

```
int matrix[4][4]={{1,0,0,0},{0,1,0,0},  
{0,0,1,0}, {0,0,0,1}}
```

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Lista di inizializzazione per matrici (3/3)

Esempio:

```
//Array 2X3
//La prima sottolista inizializza la riga di indice 0 con i valori 1,2, e 3
//La seconda sottolista inizia la riga di indice 1 con i valori 5, 6, e 7
int A[2][3] ={{1,2,3},{4,5,6}};

//Array 2X3
//La prima sottolista inizializza i primi due elementi della prima riga
//con i valori 1 e 2.
//Il terzo elemento della prima riga è implicitamente inizializzato a zero.
//La seconda sottolista inizializza esplicitamente
//il primo elemento della seconda riga a 4
int B[2][3] = {{1,2},{4}};

//Array 2X3. Notazione alternativa nella lista di inizializzazione
//Le parentesi graffe vengono rimosse dalla lista degli inizializzatori
//delle singole righe. Il compilatore inizializza gli elementi della prima
//riga seguiti dagli elementi della seconda riga, ecc.
//Gli elementi che non hanno un inizializzatore esplicito sono inizializzati
//automaticamente a zero.
int C[2][3]= {1,2,3,4,5};
```

Accesso agli elementi di una matrice (1/2)

- Ad ogni elemento di un array bidimensionale è associato **un indice di riga** che va da 0 al numero di righe meno 1 ed un **indice di colonna** che va da 0 al numero di colonne meno 1.
- Si può far riferimento a uno qualunque degli elementi di un array bidimensionale fornendo il nome dell'array seguito dall'indice di riga racchiuso tra parentesi quadre a sua volta seguito dall'indice di colonna racchiuso tra parentesi quadre.

Accesso agli elementi di una matrice (2/2)

Sintassi accesso agli elementi di una matrice.

`<nome matrice>[<indice riga>] [<indice colonna>]`

- `<indice riga>` e `<indice colonna>` sono generiche espressioni numerica intere.
- L'espressione `<nome matrice>[<indice riga>] [<indice colonna>]` è un **left value**:può essere utilizzata alla stessa stregua di una variabile avente come tipo il tipo degli elementi della matrice. In particolare, può comparire come operando sinistro in un'istruzione di assegnazione e come parte di un'espressione numerica se il tipo degli elementi è numerico.

```
int D[5][8];
```

```
int i= 2;  
D[i][i*i] = i+i;
```

Memorizzazione di una matrice (1/2)

Le matrici sono memorizzate per righe in celle contigue.

Matrice A con N righe e M colonne:



- Una variabile matrice A rappresenta una variabile puntatore all'array monodimensionale corrispondente alla prima riga della matrice.
- Per ogni indice di riga $0 \leq i \leq N-1$, A[i] è un puntatore costante il cui valore è l'indirizzo di memoria del primo elemento della riga i: **le espressioni A[i] e &A[i][0] sono equivalenti**.
- La variabile A è un puntatore costante a puntatori costanti: il suo valore coincide con l'indirizzo di memoria di A[0]: le espressioni A , &A[0], e &&A[0][0] sono equivalenti.

Memorizzazione di una matrice (2/2)

Le matrici sono memorizzate per righe in celle contigue.

Matrice A con N righe e M colonne:



- Nell'aritmetica dei puntatori, l'**offset** dell'elemento $A[i][j]$ rispetto all'indirizzo di memoria di $A[0][0]$ è $M*i + j$.

$$\&A[i][j] \leftrightarrow \&A[0][0] + M*i + j \leftrightarrow A[i]+j \leftrightarrow *(A+i) + j$$

$$A[i][j] \leftrightarrow *(&A[0][0] + M*i + j) \leftrightarrow *(A[i]+j) \leftrightarrow *(*(A+i) + j)$$

Array bidimensionali e funzioni (1/5)

- Le funzioni possono avere come parametri formali variabili di qualsiasi tipo e, dunque, anche variabili matrici (array bidimensionali).
- In una funzione, la dichiarazione di un parametro formale di tipo matrice corrisponde alla dichiarazione di una variabile matrice dove è necessario specificare il numero di colonne ma non il numero di righe.

Sintassi parametro formale array monodimensionale:

<tipo elemento> <nome parametro> [][<N_Colonne>]

- **<N_Colonne>**: deve essere un'espressione numerica costante. Va dichiarato necessariamente il numero di colonne dal momento che ciò che viene passato è un puntatore ad un array di righe ed il compilatore deve conoscere la lunghezza di ogni riga (e, cioè, il numero di colonne) per consentire l'accesso agli elementi della matrice.

Array bidimensionali e funzioni (2/5)

- **Sintassi parametro formale array monodimensionale:**

<tipo elemento> <nome parametro> [][<N_Colonne>]

- Il numero di righe è irrilevante ai fini dell'aritmetica dei puntatori su **<nome parametro>** .
- L'argomento (parametro attuale) in una chiamata di funzione deve essere il **nome di una variabile matrice** i cui elementi hanno lo stesso tipo degli elementi del parametro formale.

Array bidimensionali e funzioni (3/5)

- Una variabile matrice **A** è una **variabile puntatore a puntatore**: il suo valore è l'indirizzo di memoria dell'array monodimensionale corrispondente alla prima riga di A. Il valore di A è `&A[0]`.
- Quando in una chiamata di funzione viene passato come argomento una variabile matrice **A**, viene copiato nel corrispondente parametro formale l'indirizzo di memoria di `A[0]`.
- Di conseguenza, quando la funzione chiamata modifica nel suo corpo gli elementi del parametro formale, essa modifica gli effettivi elementi della matrice passato come argomento nelle loro originarie locazioni di memoria.
- Dunque, le matrici come gli array monodimensionali **vengono passati per riferimento**.

Array bidimensionali e funzioni (4/5)

- **Sintassi parametro formale array monodimensionale:**

<tipo elemento> <nome parametro> [][<N_Colonne>]

- **Sintassi alternativa come puntatore alla prima riga:**

<tipo elemento> (* <nome parametro>) [<N_Colonne>]

BUONA NORMA: specificare nel prototipo della funzione un parametro intero che rappresenta il numero di righe della matrice.

Array bidimensionali e funzioni (5/5)

- **Sintassi alternativa come puntatore di puntatore:**

<tipo elemento> ** <nome parametro>

In questo caso, nella chiamata di funzione, deve essere passato il nome di un array monodimensionale di puntatori di lunghezza pari al numero di righe della data matrice A e il cui elemento i-esimo ha come valore l'indirizzo della riga i-esima di A e, cioè, A[i].

BUONA NORMA: adottando la notazione di puntatore a puntatore, specificare nel prototipo della funzione un parametro intero che rappresenta il numero di righe della matrice ed un altro parametro intero rappresentante il numero di colonne.

Esempio: inizializzazione matrice (1/3)

```
#define N_Colonne 4

void InizializzaMatrice(int A[][N_Colonne], unsigned int N_Righe)
{
    int i,j;
    //Ciclo esterno: itera sul
    //numero di righe
    for(i=0;i<N_Righe;i++)
    {
        //Ciclo interni: itera sul
        //numero di colonne
        for(j=0;j<N_Colonne;j++)
        {
            printf("\n Inserisci elemento A[%u,%u]: ",i,j);
            scanf("%d",&A[i][j]);
        }
    }
}

int main()
{
    unsigned int N_Righe = 4;
    int A[N_Righe][N_Colonne];
    InizializzaMatrice(A,N_Righe);
}
```

Esempio: inizializzazione matrice (2/3)

```
#define N_Colonne 4

void InizializzaMatrice2(int (* A)[N_Colonne], unsigned int N_Righe)
{
    int i,j;
    //Ciclo esterno: itera sul
    //numero di righe
    for(i=0;i<N_Righe;i++)
    {
        //Ciclo interni: itera sul
        //numero di colonne
        for(j=0;j<N_Colonne;j++)
        {
            printf("\n Inserisci elemento A[%u,%u]: ",i,j);
            scanf("%d",&A[i][j]);
        }
    }
}

int main()
{
    unsigned int N_Righe = 4;
    int A[N_Righe][N_Colonne];
    InizializzaMatrice2(A,N_Righe);
    return 0;
}
```

Esempio: inizializzazione matrice (3/3)

```
void InizializzaMatrice3(int ** A, unsigned int N_Righe,
                         unsigned int N_Colonne)
{
    int i,j;
    for(i=0;i<N_Righe;i++)
    {
        for(j=0;j<N_Colonne;j++)
        {
            printf("\n Inserisci elemento A[%u,%u]: ",i,j);
            scanf("%d", &A[i][j]);
        }
    }
}

int main()
{
    unsigned int N_Righe = 4,N_Colonne =4;
    int A[N_Righe][N_Colonne];
    //Array di puntatori ad interi
    int * B[N_Righe];
    int i;
    for(i=0;i<N_Righe;i++)
        B[i] = A[i];
    InizializzaMatrice3(B,N_Righe,N_Colonne);
    return 0;
}
```

Matrice verso array di puntatori (1/3)

Un array di puntatori è un array monodimensionale i cui elementi sono puntatori a dati di un certo tipo.

Sintassi dichiarazione array di puntatori:

<tipo elemento> * <nome array> [<lunghezza array>]

La definizione alloca un array di N puntatori ad elementi aventi tipo **<tipo elemento>** dove N è il valore di **<lunghezza array>**.

- Ogni puntatore deve essere esplicitamente istanziato in modo statico oppure tramite funzioni di libreria per l'allocazione dinamica della memoria (lo vedremo in seguito).

Matrice verso array di puntatori (2/3)

- Una matrice $A \ N \times M$ può essere vista come un'array di puntatori costanti di lunghezza pari al numero di righe. L' i -esimo elemento dell'array memorizza l'indirizzo della riga i -esima e, cioè, il valore di $A[i]$.
- I vettori di puntatori sono più generali delle matrici dal momento che elementi distinti dell'array possono puntare ad array monodimensionali (righe) aventi lunghezza diverse (**array di array di lunghezze diversa**).
- Dunque, l'importante vantaggio offerto da un array di puntatori consiste nel fatto che esso consente di avere righe di lunghezza variabile.
- L'uso più frequente dei vettori di puntatori consiste nel memorizzare stringhe (sequenze di caratteri) di diversa lunghezza.

Matrice verso array di puntatori (3/3)

Esempio:

```
//Definizione matrice di interi 10x20
int A[10][20];
//Definizione array di puntatori ad interi
// di Lunghezza 10
int * B[10];
```

- Ad esempio, **A[3][4]** e **B[3][4]** sono entrambi riferimenti ad un **int** sintatticamente corretti.
- Tuttavia, **A** è un vettore bidimensionale: per esso sono state riservate 200 locazioni di memoria contigue di ampiezza pari a quella di un **int**.
- Per **B**, invece, la definizione alloca soltanto 10 locazioni di memoria contigue di ampiezza pari a quella per memorizzare un indirizzo di memoria. Ognuno dei 10 puntatori nell'array deve essere inizializzato esplicitamente in modo statico o **dinamico**.

Esempio utilizzo matrici

Calcolo del massimo degli elementi di una matrice di interi.

```
#define N_Colonne 4
int CalcolaMassimo(int A[][]N_Colonne],  unsigned int N_Righe)
{
    int max = A[0][0];
    int i,j;
    for(i=0;i<N_Righe;i++)
    {
        for(j=0;j<N_Colonne;j++)
        {
            if(max<A[i][j])
                max=A[i][j];
        }
    }
    return max;
}

int main()
{
    unsigned int N_Righe = 4;
    int A[N_Righe][N_Colonne];
    InizializzaMatrice(A,N_Righe);
    int max= CalcolaMassimo(A,N_Righe);
    printf("Il massimo degli elementi della matrice e\' %d",max);
    return 0;
}
```

Tipi carattere

Ricordiamo che i tipi carattere in C sono tipi interi a 1 byte utilizzati per rappresentare i caratteri alfanumerici e alcuni caratteri speciali (**sequenze di escape**) in accordo allo standard di codifica dei caratteri ASCII (American Standard Code for Information Interchange) tramiti interi a 8 bits. Vi sono due tipi per caratteri in C, uno con segno e l'altro senza segno:

- **char**
- **unsigned char**

Costante carattere: simbolo di carattere racchiuso tra apici singoli come ‘a’ . Il valore di una costante carattere è il valore numerico di quel carattere all’interno del set di caratteri della macchina (usualmente il codice ASCII del carattere).

Stringhe

- Ricordiamo che nel linguaggio C, una **costante stringa** o **stringa letterale** è una arbitraria sequenza di caratteri racchiusa tra doppi apici, ad esempio “*Esame di Programmazione*”.
- Per stringa si intende un array (monodimensionale) di caratteri (elementi di tipo **char**) il cui ultimo carattere coincide con il **carattere nullo di terminazione** ('\0'). La lunghezza di una stringa coincide con la lunghezza dell'array meno 1 (il carattere di terminazione non viene considerato).
- Il linguaggio C fornisce alcuni costrutti per il trattamento specifico di stringhe insieme a funzioni di libreria per la manipolazione di stringhe.

Dichiarazione e inizializzazione di stringhe (1/2)

Una stringa può essere dichiarata come un normale array monodimensionale di elementi di tipo **char** assicurando nella fase di inizializzazione che l'ultimo carattere sia '\0'. Il linguaggio fornisce costrutti alternativi per la dichiarazione e l'inizializzazione di variabili stringhe.

Sintassi alternativa con lista di inizializzazione:

char <nome stringa> [] = {<c_1>, ..., <c_N>, '\0'};

- <c_1>, ..., <c_N> sono costanti carattere.
- Definisce un array di caratteri di lunghezza N+1 che termina con il carattere di terminazione nulla.

Esempio:

```
char color[] = {'b', 'l', 'u', '\0'};
```

Dichiarazione e inizializzazione di stringhe (2/2)

Sintassi alternativa con inizializzazione basata su costante stringa:

char <nome stringa> [] = <costante stringa>;

oppure come un puntatore a carattere:

char * <nome stringa> = <costante stringa>;

Definisce un array di caratteri di lunghezza N+1 che termina con il carattere di terminazione nullo '\0' dove N è la lunghezza della costante stringa.

Esempio:

```
char color[] = "blue";
char * colorPtr ="blue";
```

Stampa di una stringa di caratteri

Un array di caratteri che rappresenta una stringa può essere inviato in uscita con **printf** usando lo specificatore di conversione **%s**.

Esempio:

```
int main()
{
    char color[] = "blue";
    printf("s%\n",color);
}
```

La funzione **printf** *non* controlla quanto è grande l'array di caratteri. I caratteri della stringa sono stampati finché non si incontra un carattere nullo di terminazione.

Inizializzazione di stringhe tramite scanf (1/2)

- È possibile utilizzare lo specificatore di conversione `%s` per inizializzare tramite la funzione **scanf** un array di caratteri ad una stringa di caratteri inseriti dall'utente.
- Quando **scanf** incontra lo specificatore `%s`, legge i caratteri e li memorizza nell'array di input finchè non incontra uno spazio, una tabulazione, un newline o un indicatore di fine file. A questo punto, la funzione inizializza l'elemento corrente dell'array di input con `'\0'`.

Inizializzazione di stringhe tramite scanf (2/2)

- È importante assicurarsi che il numero di caratteri processati + il carattere nullo di terminazione non superi la lunghezza del vettore di caratteri (altrimenti si genera un **overflow del buffer**). Ciò può essere garantito utilizzando lo specificatore di conversione per stringhe nel formato %Ns dove N è una costante intera non negativa che indica il numero massimo di caratteri che possono essere letti ed inseriti nell'array di input.

Esempio:

```
int main()
{
    char stringa[20];
    scanf("%19s\n",stringa);
    printf("Stringa = %s",stringa);
}
```

Libreria per il trattamento di caratteri (1/3)

La libreria di funzioni per il trattamento di caratteri (`<ctype.h>`) includono diverse funzioni che eseguono test e conversioni di dati di tipo carattere. L'argomento di ognuna di queste funzioni è un **int**, il cui valore dev'essere una quantità rappresentabile come **unsigned char**. Ognuna di queste funzioni restituiscono un **int**.

Funzioni di test: ritornano un valore diverso da zero (true) se l'argomento soddisfa la condizione descritta, zero altrimenti.

- **int isalnum (int c):** testa se c è una cifra decimale o una lettera.
- **int isalpha (int c):** testa se c è una lettera.
- **int isblank (int c):** testa se c è un *carattere della classe blank*: uno spazio o tab orizzontale ('\t').

Libreria per il trattamento di caratteri (2/3)

- **int iscntrl (int c)**: testa se c è un *carattere di controllo*: tab orizzontale ('\t'), tab verticale ('\v'), avanzamento pagina ('\f'), avviso ('\a'), backspace ('\b'), ritorno a capo ('\r'), newline ('\n').
- **int isdigit (int c)**: testa se c è una cifra decimale.
- **int isgraph (int c)**: testa se c è un carattere stampabile diverso da uno spazio.
- **int islower (int c)**: testa se c è una lettera minuscola.
- **int isprint (int c)**: testa se c è un carattere stampabile incluso uno spazio.
- **int ispunct (int c)**: testa se c è un carattere stampabile diverso da uno spazio, da una cifra o da una lettera.

Libreria per il trattamento di caratteri (3/3)

- **int isspace (int c)**: testa se c è un *carattere di spaziatura*: newline ('\n'), spazio (' '), avanzamento pagina ('\f'), ritorno a capo ('\r'), tab orizzontale ('\t') o verticale ('\v').
- **int isupper (int c)**: testa se c è una lettera maiuscola.
- **int isxdigit (int c)**: testa se c è carattere esadecimale.

Funzioni di conversione:

- **int tolower (int c)**: se c è una lettera maiuscola, restituisce c come lettera minuscola. Altrimenti, restituisce l'argomento inalterato.
- **int toupper (int c)**: se c è una lettera minuscola, restituisce c come lettera maiuscola. Altrimenti, restituisce l'argomento inalterato.

Esempio gestione caratteri

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

//Converti una stringa in lettere maiuscole
void ConvertiLettereMaiuscole(char * sPtr)
{
    while(*sPtr != '\0')//IL carattere corrente non è '\0'
    {
        *sPtr = toupper(*sPtr);//Converti in maiuscolo
        ++sPtr;//fai puntare sPtr al carattere successivo
    }
}

//Stampa i singoli caratteri di una stringa
//Il parametro d'input è un puntatore a dati costanti
//dal momento che la stringa è solo acceduta in lettura
void StampaCaratteriStringa(const char *sPtr)
{
    for(;*sPtr != '\0'; ++sPtr)//nessuna inizializzazione
        printf("%c",*sPtr);
}

int main()
{
    char str[] = "Stampami In Lettere Maiuscole";
    ConvertiLettereMaiuscole(str);
    StampaCaratteriStringa(str);
}
```

Libreria per il trattamento di stringhe

La **libreria di funzioni per il trattamento di stringhe** (`<string.h>`) fornisce diverse funzioni per:

- Manipolazione di stringhe: copia e concatenazione.
- Confronto lessicografico tra stringhe.
- Ricerca di sotto-stringhe all'interno di stringhe.
- Suddivisione di stringhe in **token**.
- Determinazione della lunghezza di una stringa.
- Funzioni di gestione della memoria (lo vedremo in seguito).

Le funzioni della libreria utilizzano la macro **size_t** che denota un tipo intero senza segno (dipendente dal compilatore).

Copia e concatenazione di stringhe (1/2)

- **char * strcpy (char * s1, const char * s2)**: copia il suo secondo argomento **s2** (array di caratteri) nel suo primo argomento **s1** e restituisce il valore di **s1**. È necessario assicurarsi che l'array di caratteri **s1** sia grande abbastanza per memorizzare la stringa (incluso il carattere nullo di terminazione) riferita da **s2**.
- **char * strncpy (char * s1, const char * s2, size_t n)**: equivalente a **strcpy**, ma **strncpy** addizionalmente specifica il numero di caratteri da copiare nel primo argomento **s1**. Questo implica che la funzione non copia il carattere nullo di terminazione se il parametro **n** è minore della lunghezza dell'array **s2**.

Copia e concatenazione di stringhe (2/2)

- **char * strcat (char * s1, const char * s2):** aggiunge il suo secondo argomento **s2** (stringa) in coda al suo primo argomento **s1** e restituisce il valore di **s1**. Il primo carattere del secondo argomento sostituisce il carattere '\0' che termina la stringa nel primo argomento. È necessario assicurarsi che l'array di caratteri **s1** sia grande abbastanza per memorizzare la prima stringa concatenata con la seconda stringa.
- **char * strncat (char * s1, const char * s2, size_t n):** aggiunge un numero specificato **n** di caratteri della seconda stringa in coda alla prima e restituisce il valore di **s1**. Un carattere nullo di terminazione è aggiunto in coda automaticamente al risultato.

Esempio manipolazione stringhe

```
#include <string.h>

int main()
{
    char x[] = "Verra\' l'ora del giudizio e la verita\' risplendera\'!";
    char y[] = "la verita\' risplendera\'!";
    size_t SIZE = 100;
    char z[SIZE];
    strncpy(z,x,28);
    z[28] = '\0';
    printf("La stringa z e': %s\n",z);
    printf("Ora la stringa z e': %s\n",strcat(z,y));
}
```

```
La stringa z e': Verra' l'ora del giudizio e
Ora la stringa z e': Verra' l'ora del giudizio e la verita' risplendera'!
```

Confronto tra stringhe

- **char * strcmp (char * s1, const char * s2)**: confronta la stringa **s1** con la stringa **s2**, restituendo 0, un valore minore di 0 o maggiore di 0 se **s1** è, rispettivamente, uguale, minore o maggiore di **s2** rispetto all'ordinamento lessicografico basato sui codici numerici (ASCII o Unicode) dei singoli caratteri che compongono una stringa.
- **char * strncmp (char * s1, const char * s2, size_t n)**: confronta fino a **n** caratteri della stringa **s1** con la stringa **s2**. La funzione restituisce 0, un valore minore di 0 o maggiore di 0 se **s1** è, rispettivamente, uguale, minore o maggiore di **s2** per i primi **n** caratteri.

Ricerca di caratteri e sottostringhe (1/2)

- **char * strchr (const char * s1, int c)**: cerca *la prima* occorrenza del carattere **c** nella stringa **s1**. Se il carattere viene trovato, restituisce un puntatore al carattere in **s1**. Altrimenti, restituisce **NULL**.
- **char * strrchr (const char * s1, int c)**: cerca *l'ultima* occorrenza del carattere **c** nella stringa **s1**. Se il carattere viene trovato, restituisce un puntatore al carattere in **s1**. Altrimenti, restituisce **NULL**.
- **Size_t strncspn (const char * s1, const char * s2)**: restituisce la lunghezza della *parte iniziale* della stringa nel suo primo argomento che **non** contiene alcun carattere appartenente alla stringa nel suo secondo argomento.

Ricerca di caratteri e sottostringhe (2/2)

- **Size_t strnspn (const char * s1, const char * s2)**: restituisce la lunghezza della *parte iniziale* della stringa nel suo primo argomento che contiene solo caratteri della stringa nel suo secondo argomento.
- **char * strpbrk (const char * s1, const char * s2)**: cerca nel primo argomento **s1** (una stringa) *la prima occorrenza* di un qualsiasi carattere che fa parte del suo secondo argomento **s2** (un'altra stringa). Se un carattere viene trovato, restituisce un puntatore al carattere in **s1**. Altrimenti, restituisce NULL.
- **char * strpstr (const char * s1, const char * s2)**: se la prima stringa **s1** contiene una sottostringa uguale a **s2**, restituisce un puntatore all'inizio della prima occorrenza di tale sottostringa in **s1**. Altrimenti, restituisce NULL.

Ricerca di token in stringhe

char * strtok (char * s1, const char * s2)

- Suddivide la stringa **s1** in una collezione di token, sottostringhe della stringa **s1** separate da caratteri (delimitatori) contenuti nella stringa **s2**. Ad esempio, in una riga di testo, ogni parola può essere considerata un token e gli spazi e la punteggiatura che separano le parole possono essere considerati delimitatori.
- Sono necessarie più chiamate alla funzione per ottenere i token. Nella prima chiamata, con il primo argomento che rappresenta la stringa data, viene restituito un puntatore in **s1** al primo token ed il primo delimitatore in **s1** viene sostituito con ‘\0’. Nelle chiamate successive, vengono restituiti i puntatori agli altri token in **s1** fino a quando non viene restituito NULL (indica che non ci sono più token). In queste chiamate secondarie, il primo argomento deve essere NULL. La funzione tiene traccia internamente, tramite una variabile statica, di un puntatore al carattere successivo della stringa data.

Esempio suddivisione in token

```
#include <string.h>

int main()
{
    char str[] = "Questa e\' una frase con 7 token!";
    //Usiamo lo spazio come delimitatore
    //Prima chiamata a strtok
    char * tokenPtr = strtok(str, " ");

    while(tokenPtr != NULL)
    {
        printf("%s\n",tokenPtr);
        //Successive chiamate a strtok
        tokenPtr = strtok(NULL, " ");
    }
}
```

OUTPUT

```
Questa
e'
una
frase
con
7
token!
```

Lunghezza di una stringa

size_t strlen (const char * s): determina la lunghezza della stringa s restituendo il numero di caratteri in s che precedono il carattere nullo '\0' di terminazione.

Esempio:

```
#include <string.h>

int main()
{
    const char str[] = "Questa e' una frase con 7 token!";
    printf("La lunghezza di ''%s'' e':\n      %u",str,strlen(str));
}
```

OUTPUT

```
La lunghezza di ''Questa e' una frase con 7 token!'' e':
      32
```

Lezione 16: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi su Matrici

1. Scrivere una funzione per l'inizializzazione di una matrice di interi, un'altra funzione per la stampa a video degli elementi di una matrice di interi, ed una funzione per la ricerca di un elemento all'interno di una matrice che restituisca un'indicazione dell'esito della ricerca (elemento trovato o meno). Testare le tre funzioni nel main.
2. Scrivere una funzione che prenda in input due matrici aventi le stesse dimensioni (e, cioè, lo stesso numero di righe e colonne) e copi il contenuto della prima matrice nella seconda. Testare la funzione utilizzando le funzioni dell'esercizio 1 per l'inizializzazione della prima matrice e la stampa degli elementi della seconda matrice.

Esercizi su Matrici

3. Una matrice si dice **quadrata** se il numero di colonne coincide con il numero di righe. Data una matrice quadrata $A \in \mathbb{N} \times \mathbb{N}$, si definisce **trasposta di A**, la matrice quadrata $A^T \in \mathbb{N} \times \mathbb{N}$ definita come segue: $A^T[i][j]$ coincide con $A[j][i]$ per tutti $0 \leq i, j \leq N-1$ (matrice ottenuta scambiando le righe con le colonne). Scrivere una funzione che data in input una matrice quadrata A di interi converta A nella sua trasposta (senza utilizzare matrici di supporto). Testare la funzione utilizzando le funzioni dell'esercizio 1 per l'inizializzazione di una matrice e la stampa degli elementi di una matrice.
4. Scrivere una funzione che prenda in input una matrice A di float ed un array (monodimensionale) di float la cui lunghezza coincida con il numero di righe di A . Per ogni indice di riga i , la funzione deve aggiornare l'elemento di indice i dell'array con la somma degli elementi della riga i -esima di A . Testare la funzione utilizzando la funzione dell'esercizio 1 per l'inizializzazione di una matrice ed una funzione per la stampa degli elementi di un array monodimensionale.

Esercizi su Matrici

5. Una matrice quadrata \mathbf{A} $N \times N$, si definisce **simmetrica** se $\mathbf{A}[i][h] = \mathbf{A}[h][i]$ per tutti $0 \leq i, h \leq N-1$. Scrivere una funzione che data in input una matrice quadrata \mathbf{A} di interi restituendo un intero indicante l'esito del test. Testare la funzione utilizzando la funzione dell'esercizio 1 per l'inizializzazione di una matrice.

Simmetrica

1	4	21	-8
4	3	-6	2
21	-6	7	1
-8	2	1	-5

Non Simmetrica

1	4	21	5
4	3	-6	2
21	-6	7	1
-8	2	1	-5

Esercizi su Matrici

6. Scrivere una funzione che prenda in input una matrice di float ed utilizzando una funzione ricorsiva che implementi l'algoritmo **Merge Sort** ordini per valori crescenti gli elementi di ciascuna riga della matrice (senza utilizzare matrici di supporto). Testare la funzione utilizzando le funzioni dell'esercizio 1 per l'inizializzazione di una matrice e la stampa degli elementi di una matrice.

Esercizi su stringhe

Nei seguenti esercizi utilizzare la funzione **gets** della libreria standard di input ed output del linguaggio C. Il prototipo della funzione **gets** è

```
char *gets(char *s);
```

Il prototipo è definito in stdio.h. La funzione legge una linea dallo standard input fino al carattere di new-line '\n' o di EOF (fine file). Questi caratteri sono sostituiti con '\0'.

ATTENZIONE: non viene eseguito nessun controllo sulla lunghezza dell'array puntato da **s**, pertanto se la linea da leggere supera la lunghezza dell'array si ha un **buffer overflow**. La funzione ritorna **s** in caso di successo e **NULL** quando nessun carattere è letto prima del carattere di new-line '\n' o di EOF.

Esercizi su stringhe

7. Scrivere una funzione che prenda in input una stringa e restituisca la lunghezza della stringa (e cioè il numero di caratteri prima del carattere di terminazione '\0'). La funzione non deve utilizzare funzioni di libreria. Testare la funzione nel main tramite la funzione di libreria **gets** per l'inserimento di una stringa da tastiera.

8. Scrivere una funzione che prenda in input una stringa e converta le lettere nella data stringa in lettere maiuscole (utilizzando la funzione **toupper** in **string.h**). Testare la funzione nel main inizializzando una stringa da tastiera con **gets** e stampando la stringa finale.

Esercizi su stringhe

9. Scrivere una funzione che prenda in input una stringa e converta la stringa nella sua inversa (e, cioè la stringa originaria letta da destra verso sinistra) senza utilizzare array di supporto. Testare la funzione nel main inizializzando una stringa da tastiera con **gets** e stampando la stringa finale.
10. Scrivere una funzione che prenda in input una stringa e restituisca il numero di parole contenute nella stringa dove una parola contiene solo lettere e cifre decimali. Utilizzare la funzione **isalnum**, il cui prototipo è in ctype.h, per testare se un carattere è o una lettera o una cifra decimale. Testare la funzione nel main inizializzando una stringa da tastiera con **gets**.

Esercizi su stringhe

11. Scrivere una funzione che prenda in input una stringa e ritorni come parametri di output (parametri definiti come puntatori) il carattere che occorre il maggior numero di volte nella stringa ed il suo numero di occorrenze. Se vi sono più caratteri che occorrono il maggior numero di volte, scegliere il primo che occorre nella stringa data. Testare la funzione nel main inizializzando una stringa da tastiera con **gets**.

Lezione 16

Antonio Origlia
a.a. 2022/2023

Sommario - Lezione 17: Strutture dati dinamiche (1)

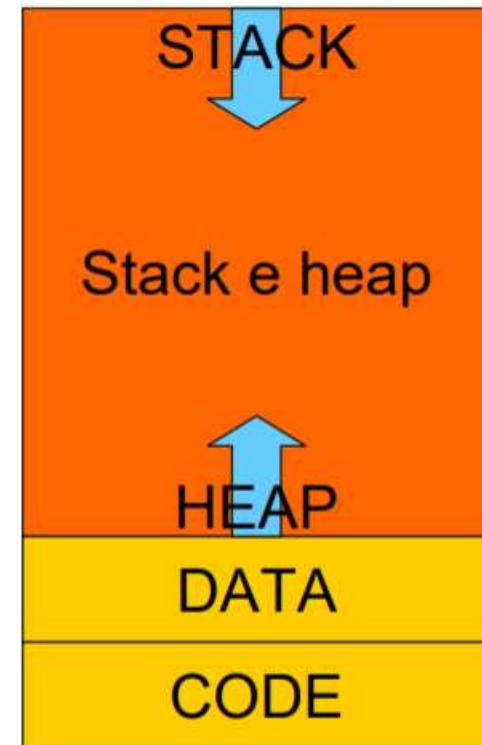
- Allocazione della memoria.
- Allocazione dinamica della memoria: la funzione malloc.
- Rilascio della memoria dinamica: la funzione free.
- Allocazione dinamica di matrici.
- Strutture autoreferenziali.
- Introduzione alle liste concatenate.
- Differenze tra liste e array.
- Implementazione delle principali operazioni su liste.

Allocazione della memoria (1/2)

- Il termine **allocazione** viene utilizzato per indicare l'assegnazione di un blocco di memoria RAM ad un programma.
- La memoria per i dati può essere allocata nello **stack** di esecuzione del programma, nell'**heap** (detto anche **memoria dinamica**) o nel **Data Segment** di un programma.
- La memoria può essere allocata:
 - A run-time (dal programma in esecuzione)
 - A compile-time (dal compilatore)
- La memoria può essere rilasciata (deallocata, resa disponibile) automaticamente o a richiesta.

Allocazione della memoria (2/2)

- Il programma compilato è costituito da due parti distinte:
 - **Code Segment**
 - **Data Segment**
- Quando il programma viene eseguito, il Sistema Operativo alloca uno spazio di memoria per lo **stack** e l'**heap**.
- Lo spazio di memoria per lo stack può esaurirsi per effetto di ripetute chiamate annidate a funzione.
- Lo spazio di memoria per l'heap può esaurirsi per ripetute allocazioni dinamiche.



Allocazione automatica

- Si ha **allocazione automatica** di un blocco di memoria quando in una funzione (main incluso) viene definita una variabile locale (senza lo specificatore di classe di memoria **static**).
- Il blocco di memoria per la variabile locale automatica viene creato a run-time nello **stack** di esecuzione del programma all'interno del **record di attivazione** allocato per gestire un'attivazione dell'esecuzione della specifica funzione.
- Il blocco di memoria per la variabile locale automatica **non è rilasciabile esplicitamente**, ma avviene automaticamente quando termina l'esecuzione della funzione dove è definita la variabile.

Allocazione statica

- Si ha **allocazione statica** di un blocco di memoria quando viene definita una variabile di classe di allocazione statica e, cioè,
 - o una variabile esterna (globale) definita al di fuori di una funzione,
 - o una variabile locale definita all'interno di una funzione premettendo lo specificatore di classe di memoria **static**.
- Il blocco di memoria viene allocato a **compile-time** nel **Data Segment**.
- Il blocco di memoria non è rilasciabile (non è riutilizzabile per altre allocazioni).

Variabili definite in una funzione

```
void Incrementa()
{
    int x=0;
    x++;
    printf("%d\n",x);
}

int main()
{
    Incrementa();
    Incrementa();
    Incrementa();|
```

Produce come output:
1
1
1

```
void Incrementa()
{
    static int x=0;
    x++;
    printf("%d\n",x);
}

int main()
{
    Incrementa();
    Incrementa();
    Incrementa();|
```

Produce come output:
1
2
3

Allocazione dinamica (1/2)

- Per l'**allocazione dinamica** di un blocco di memoria si usano opportune funzioni di libreria.
- Il blocco di memoria viene allocato a **run-time** nell'**heap**.
- L'ampiezza del blocco di memoria da allocare dinamicamente è specificata come parametro d'input nella chiamata della funzione di allocazione.
- Il blocco di memoria è **rilasciabile** (per poter essere utilizzato per altre allocazioni dinamiche) solo **esplicitamente** per mezzo di opportune funzioni di libreria (non è automatico).

Allocazione dinamica (2/2)

- Il funzionamento dell'heap è gestito dal sistema operativo. Le zone di memoria nell'**heap** sono marcate libere o occupate.
- L'allocazione dinamica di memoria è fondamentale nella gestione di **strutture dati dinamiche** e cioè strutture dati la cui ampiezza può aumentare e diminuire durante l'esecuzione: ad esempio, liste concatenate, code, pile, ed alberi.
- **Gestione puntatori:** un puntatore contiene l'indirizzo di memoria di un blocco di memoria allocato. Finora abbiamo visto come assegnare ad un puntatore l'indirizzo di una delle variabili del programma (dati primitivi, strutture, unioni, o array).
 - **Metodo alternativo di gestione dei puntatori:** allocazione dinamica della memoria attraverso una chiamata di funzione che alloca un blocco di memoria e ne restituisce l'indirizzo iniziale.

Funzione malloc (1/4)

Prototipo della funzione (dichiarato in **stdlib.h**):

```
void * malloc(size_t size);
```

- In caso di successo, alloca un blocco di byte contiguo non inizializzato nell'**heap** avente un numero di byte pari a **size** e restituisce un puntatore di tipo **void** il cui valore è l'indirizzo di memoria del primo byte del blocco allocato.
- In caso di insuccesso (il blocco di memoria della specifica ampiezza non può essere allocato perché non c'è sufficiente spazio contiguo libero nell'heap), viene restituito il valore **NULL**.

BUONA NORMA: controllare se il valore di ritorno è diverso da **NULL** (e, cioè, il blocco di memoria è stato allocato con successo).

Funzione malloc (2/4)

Viene di solito utilizzata in congiunzione con l'operatore **sizeof** per specificare l'ampiezza di un tipo di dato che si vuole allocare dinamicamente.

Esempio:

```
#include <stdlib.h>
...
int * p = (int *) malloc(sizeof(int));
if( p!= null)
{
    *p = 24;
    (*p)++;
}
```

- attivando **malloc(sizeof(int))** viene allocata una zona di memoria adatta a contenere un intero **int**; ovvero viene creata una nuova variabile intera.
- il puntatore restituito da **malloc** viene assegnato al puntatore **p** (dopo un cast esplicito): dopo l'assegnazione **p** contiene l'indirizzo iniziale dell'intero allocato.

Funzione malloc (3/4)

- Il blocco di byte allocato con **malloc** non ha di per sé alcun tipo: il cast sul puntatore restituito fa sì che il blocco di byte sia considerato dal compilatore come avente il tipo indicato nel cast.
- Nell'esempio precedente, il cast (**int ***) fa sì che il compilatore consideri il blocco di byte come vettore di interi (nel caso specifico abbiamo un vettore di interi di lunghezza 1 per gestire una variabile di tipo **int**).
- Non si può applicare l'operatore **sizeof** a un blocco di memoria allocato dinamicamente in quanto **sizeof** viene valutato dal compilatore.

Funzione malloc (4/4)

- Il cast esplicito non sarebbe necessario per un compilatore C standard, perché l'assegnazione di un puntatore **void** ad un puntatore **non-void** non lo richiede necessariamente.
- Alcuni compilatori (in particolare quelli che compilano anche codice C++) lo richiedono comunque e quindi è bene aggiungerlo, anche per chiarezza e documentazione.

Rilascio della memoria dinamica

- Nel linguaggio C, la memoria allocata dinamicamente non viene rilasciata automaticamente (ovviamente questo capita comunque quando il programma termina in quanto tutta la memoria associata al programma viene rilasciata).
- Il C non ha un **garbage collector** che “recupera” a run-time la memoria inutilizzata.
- A run-time la memoria dinamica può essere rilasciata solo in modo esplicito (tramite la funzione di libreria **free**): questo permette di riutilizzare quella porzione di memoria per servire successive allocazioni dinamiche.

Funzione free (1/3)

Prototipo della funzione (dichiarato in **stdlib.h**):

void free(void * p);

- Il puntatore d'input **p** deve puntare all'inizio di un blocco di memoria precedentemente allocato dinamicamente.
- La funzione rilascia il blocco di memoria puntato dal puntatore **p** (il sistema operativo mantiene traccia del numero di byte che era stata allocata).

Esempio: `#include <stdlib.h>`

```
...
int * p = (int *) malloc(sizeof(int));
...
...
// Alcuni compilatori (in particolare i compilatori
// C++) richiedono un cast di p a (void *).
free((void *)p);
```

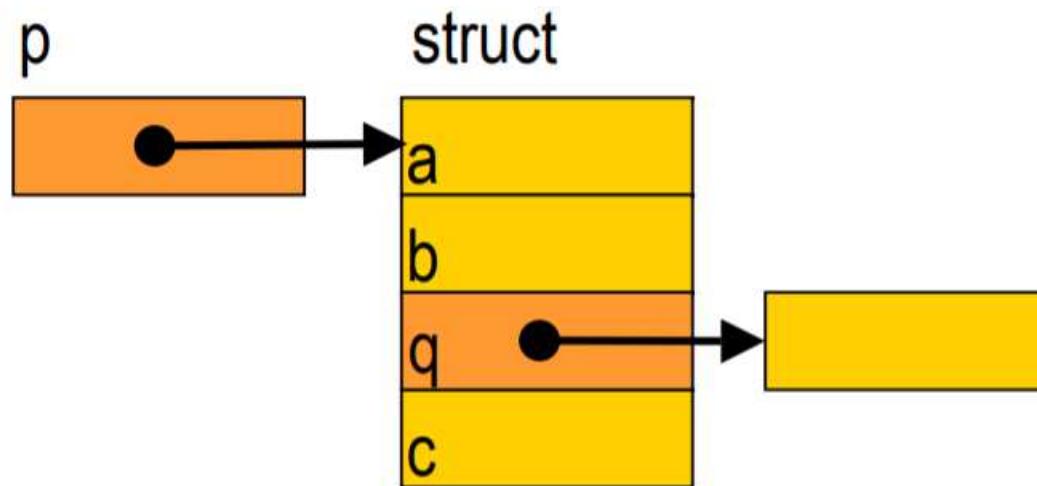
Funzione free (2/3)

- Quando la memoria che è stata allocata dinamicamente non è più necessaria, si utilizza **free** per “ restituire ” immediatamente la memoria al sistema operativo.
- Liberare la memoria non allocata in modo dinamico è un errore.
- Fare riferimento alla memoria che è stata liberata è un errore che provoca tipicamente l’arresto anomalo del programma.
- Una volta chiamata **free** per rilasciare un blocco di memoria, i valori dei puntatori al blocco dovrebbero essere impostati a NULL per eliminare la possibilità che il programma faccia riferimento alla memoria che è stata liberata e che potrebbe già essere stata allocata per un altro scopo.

Funzione free (3/3)

- Se un puntatore **p** punta ad una variabile dinamica di tipo **struct** contenente come campo (membro) un puntatore **q** ad un'altra variabile dinamica, bisogna rilasciare prima **q** e poi **p**:

```
free(p->q);  
free(p);
```



Esempi di allocazione dinamica (1/2)

Allocazione dinamica di strutture.

```
struct Punto
{
    float x; //coordinata Lungo l'asse X
    float y; //coordinata Lungo l'asse Y
};

struct Punto * AllocaInizializzaPunto()
{
    struct Punto * p= (struct Punto *)malloc(sizeof(struct Punto));
    if(p != NULL)
    {
        printf("Inserisci ascissa x: ");
        scanf("%f",& p->x);
        printf("Inserisci ordinata y: ");
        scanf("%f",& p->y);
    }
    return p;
}

void main()
{
    struct Punto * p = AllocaInizializzaPunto();
    if(p != NULL)
        printf("Punto di ascissa %.2f e ordinata %.2f",p->x,p->y);
}
```

Esempi di allocazione dinamica (2/2)

Allocazione dinamica di vettore di strutture.

```
struct Punto * AllocaPunti(size_t numPunti)
{
    struct Punto * p= (struct Punto *)malloc(numPunti * sizeof(struct Punto));
    return p;
}

void InizializzaPunti(struct Punto * pPunti,size_t numPunti)
{
    int i;
    for(i=0;i<numPunti;i++)
    {
        printf("Inserisci ascissa del punto %d: ",i+1);
        scanf("%f", &pPunti[i].x);
        printf("Inserisci ordinata del punto %d: ",i+1);
        scanf("%f", &pPunti[i].y);
    }
}
void main()
{
    struct Punto * p = AllocaPunti(5);
    if(p != NULL)
    {
        InizializzaPunti(p,5);
        int i;
        for(i=0;i<5;i++)
            printf("Punto %d di ascissa %.2f e ordinata %.2f\n",i+1, p[i].x,p[i].y);
    }
}
```

Allocazione dinamica di matrici

Vi sono essenzialmente due modi:

- **Allocazione dinamica con memoria non-contigua.**
- **Allocazione dinamica con memoria contigua.**

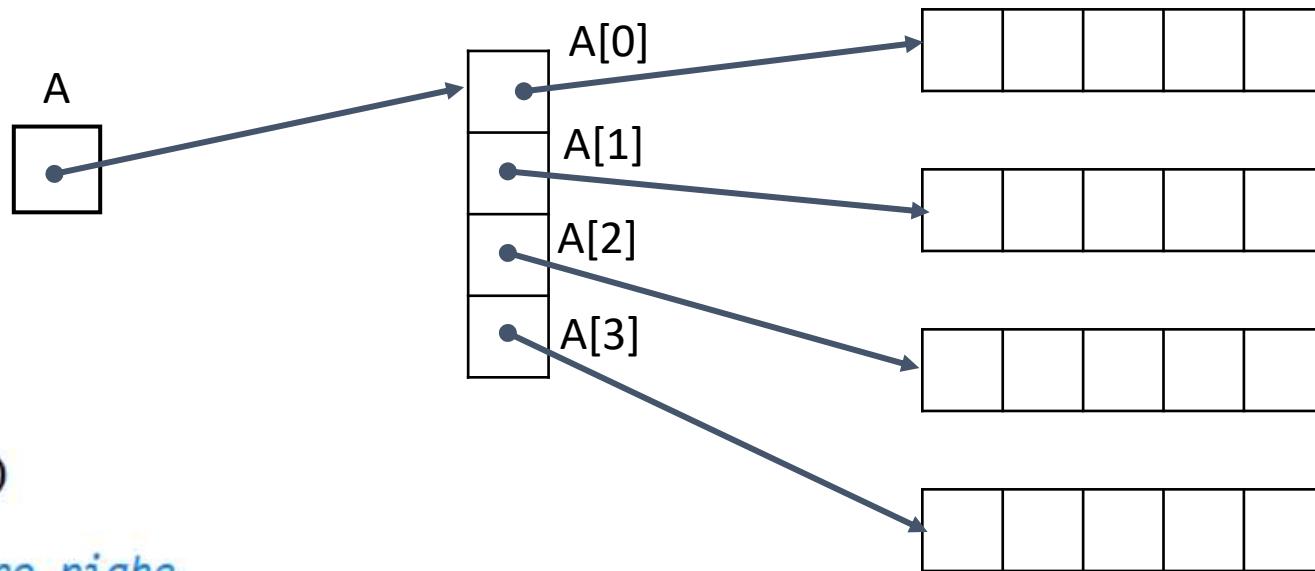
Allocazione dinamica non-contigua di matrici (1/4)

Allocazione dinamica non-contigua per matrici di elementi di tipo **<tipo>** avente **<N_R>** righe e **<N_C>** colonne (**<N_R>** e **<N_C>** sono generiche espressioni numeriche).

- **Primo step:** si alloca dinamicamente un array monodimensionale di puntatori a **<tipo>** avente lunghezza pari al numero **<N_R>** di righe (**array di puntatori alle righe da allocare**):
<tipo> ** <A> = (<tipo> **) malloc(<N_R> * sizeof(<tipo>));
- **Secondo step:** si inizializza il valore (indirizzo di memoria) dell'elemento i-esimo **<A>[i]** dell'array **<A>** di puntatori all'indirizzo iniziale di un'array monodimensionale di lunghezza **<N_C>** allocato dinamicamente:

```
for(int i=0;i < <N_R>; i++)
    <A>[i] = (<tipo> *) malloc(<N_C> * sizeof(<tipo>));
```

Allocazione dinamica non-contigua di matrici (2/4)



```
void main()
{
    //Numero righe
    int N=4;
    //Numero colonne
    int M=5;

    int ** A = (int**) malloc(N*sizeof(int *));
    int i;
    for(i=0;i<N;i++)
        A[i] = (int *) malloc(M*sizeof(int));
}
```

Allocazione dinamica non-contigua di matrici (3/4)

```
void main()
{
    //Numero righe
    int N=4;
    //Numero colonne
    int M=5;

    int ** A = (int**) malloc(N*sizeof(int *));
    int i;
    for(i=0;i<N;i++)
        A[i] = (int *) malloc(M*sizeof(int));
}
```

- **Accesso agli elementi:** come per le matrici dichiarate staticamente.

```
A[i][j] = 12;
```

- **Parametro in chiamata di funzione:** come puntatore di puntatore.

```
void f(int ** B, int N_R, int N_C);
...
f(A,N,M);
```

Allocazione dinamica non-contigua di matrici (4/4)

```
void main()
{
    //Numero righe
    int N=4;
    //Numero colonne
    int M=5;

    int ** A = (int** )malloc(N*sizeof(int *));
    int i;
    for(i=0;i<N;i++)
        A[i] = (int *) malloc(M*sizeof(int));
}
```

- **Rilascio memoria:** prima si rilascia la memoria di ciascuna riga e poi si rilascia l'array di puntatori alle righe.

```
int i=0;
for(i=0;i<N;i++)
    free(A[i]);

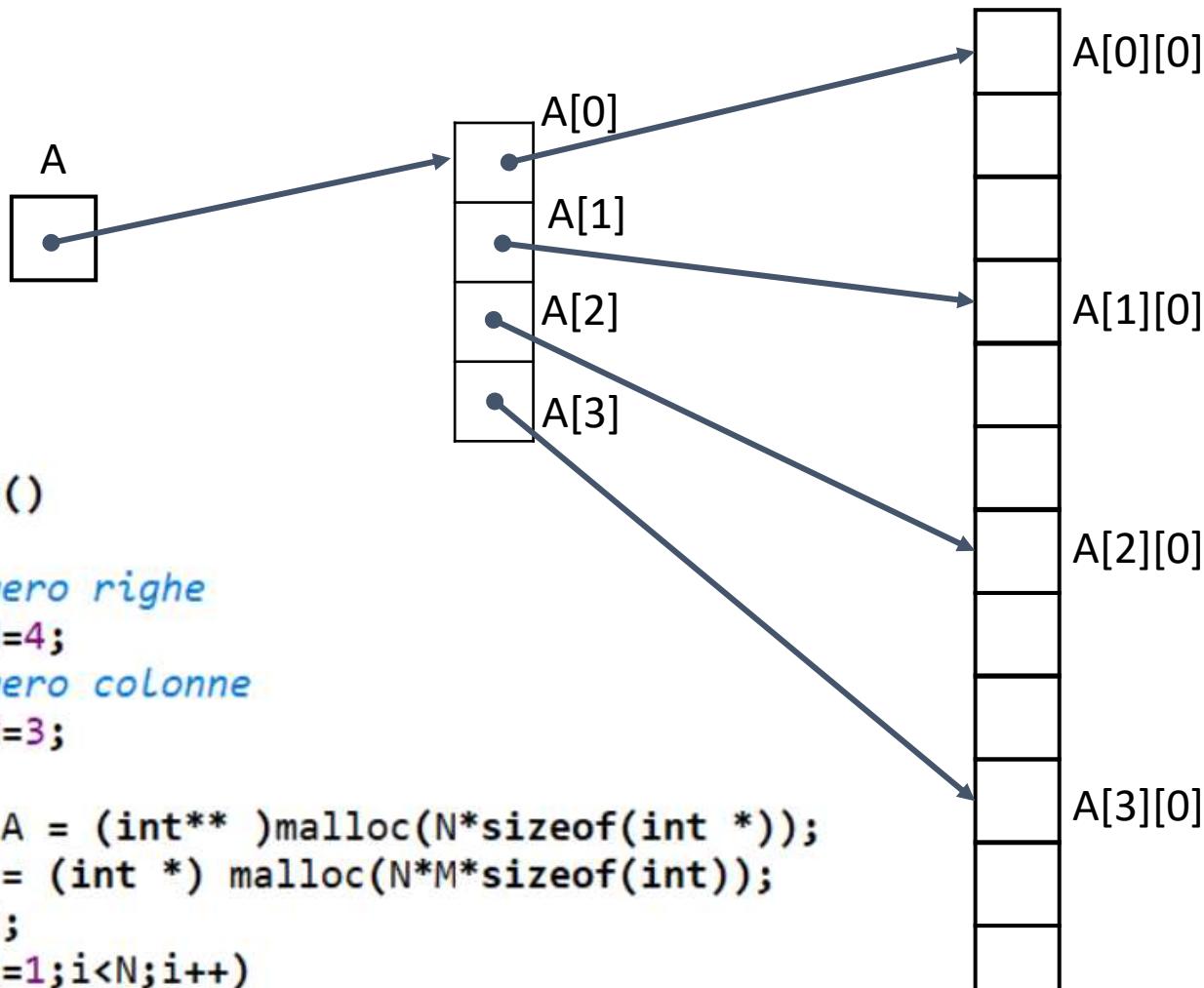
free(A);
```

Allocazione dinamica contigua di matrici (1/4)

Allocazione dinamica contigua per matrici di elementi di tipo `<tipo>` avente `<N_R>` righe e `<N_C>` colonne (`<N_R>` e `<N_C>` sono generiche espressioni numeriche).

- **Primo step:** come per il caso non-contiguo si alloca dinamicamente un array monodimensionale di puntatori alle righe da allocare:
`<tipo> ** <A> = (<tipo> **) malloc(<N_R> * sizeof(<tipo> *));`
- **Secondo step:** si inizializza il valore (indirizzo di memoria) del primo elemento `<A>[0]` dell'array `<A>` di puntatori all'indirizzo iniziale di un'array monodimensionale di lunghezza `<N_C> * <N_R>` :
`<A>[0] = (<tipo> *) malloc(<N_C> * <N_R> * sizeof(<tipo>));`
- **Terzo step:** si assegna all'elemento i-esimo `<A>[i]` dell'array `<A>` con $i \neq 0$ l'indirizzo iniziale della riga i-esima:
`for(int i=1;i < <N_R>; i++)`
`<A>[i] = A[0] + i * <N_C>;`

Allocazione dinamica contigua di matrici (2/4)



Allocazione dinamica contigua di matrici (3/4)

```
void main()
{
    //Numero righe
    int N=4;
    //Numero colonne
    int M=3;

    int ** A = (int**) malloc(N*sizeof(int *));
    A[0] = (int *) malloc(N*M*sizeof(int));
    int i;
    for(i=1;i<N;i++)
        A[i] = A[0] + i* M;
}
```

- **Accesso agli elementi:** come per le matrici dichiarate staticamente.
 $A[i][j] = 12;$
- **Parametro in chiamata di funzione:** come puntatore di puntatore.
`void f(int ** B, int N_R, int N_C);`
...
`f(A,N,M);`

Allocazione dinamica contigua di matrici (4/4)

```
void main()
{
    //Numero righe
    int N=4;
    //Numero colonne
    int M=3;

    int ** A = (int**) malloc(N*sizeof(int *));
    A[0] = (int *) malloc(N*M*sizeof(int));
    int i;
    for(i=1;i<N;i++)
        A[i] = A[0] + i* M;
}
```

- **Rilascio memoria:** prima si rilascia la memoria puntata da $A[0]$ (puntatore al primo elemento della prima riga) e poi si rilascia l’array di puntatori alle righe.

```
free(A[0]);
free(A);
```

Nota su allocazione statica di matrici (1/2)

- Una variabile matrice **A** NON allocata dinamicamente NON può essere passata come argomento in un chiamata di funzione il cui corrispondente parametro formale è un puntatore di puntatore al tipo degli elementi della matrice.
- Per consentire il passaggio come puntatore di puntatore è necessario definire un array di puntatori di lunghezza pari al numero di righe di **A** e inizializzare l'elemento i-esimo dell'array all'indirizzo iniziale **A[i]** della riga i-esima.

Nota su allocazione statica di matrici (2/2)

```
void InizializzaMatrice3(int ** A,  unsigned int N_Righe,
                         unsigned int N_Colonne)
{
    int i,j;
    for(i=0;i<N_Righe;i++)
    {
        for(j=0;j<N_Colonne;j++)
        {
            printf("\n Inserisci elemento A[%u,%u]: ",i,j);
            scanf("%d", &A[i][j]);
        }
    }
}

int main()
{
    unsigned int N_Righe = 4,N_Colonne =4;
    int A[N_Righe][N_Colonne];
    //Array di puntatori ad interi
    int * B[N_Righe];
    int i;
    for(i=0;i<N_Righe;i++)
        B[i] = A[i];
    InizializzaMatrice3(B,N_Righe,N_Colonne);
    return 0;
}
```

Strutture autoreferenziali (1/2)

- Ricordiamo che nel linguaggio C, una **struttura**, detta anche aggregato, è una collezione di dati (variabili), di uno o più tipi, raggruppate sotto un unico nome. I **campi o membri** di una struttura possono essere variabili di tipi (predefiniti o definiti dall'utente) differenti e anche **variabili puntatore**.
- Una variabile puntatore **indipendentemente dal tipo di dati a cui punta** necessita sempre dello stesso spazio di memoria per la sua allocazione e, cioè, un blocco di memoria di ampiezza sufficiente per contenere un indirizzo di memoria.
- Dall'osservazione nel punto precedente, è possibile definire tipi di **struttura autoreferenziali** aventi come **campi dei puntatori a dati del medesimo tipo**. *Le strutture autoreferenziali insieme all'allocazione dinamica della memoria rappresentano i due aspetti chiave nel linguaggio C per definire strutture dati dinamiche.*

Strutture autoreferenziali (2/2)

Esempio:

```
struct PuntoConLink{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y

    struct PuntoConLink * pPtr; //Puntatore ad un punto
};

void StampaPunto(struct PuntoConLink * pt)
{
    if(pt != NULL)
        printf("Punto con ascissa %.1f e ordinata %.1f",pt->x,pt->y);
}

void main()
{
    struct PuntoConLink pt1, pt2;
    pt1.x = pt2.x =2.3;
    pt1.x = 3;
    pt2.y = 5;
    pt1.pPtr = & pt2;
    StampaPunto(pt1.pPtr);
}
```

OUTPUT

Punto con ascissa 2.3 e ordinata 5.0

Introduzione alle liste concatenate (1/4)

I linguaggi di programmazione strutturati usualmente supportano due tipi di dati per definire sequenze lineari di dati omogenei (dello stesso tipo): **gli array** e le **liste concatenate**.

ARRAY:

- I singoli dati (elementi) in un array possono essere acceduti in modo diretto tramite l'indice di posizione.
- Elementi consecutivi in un array hanno **locazioni di memoria contigue**.
- La lunghezza di un array, e cioè, il suo numero di elementi, una volta che l'array è stato definito, non può essere modificata nel corso dell'esecuzione (**struttura dati non dinamica**).

Introduzione alle liste concatenate (2/4)

- Similmente agli array, gli elementi (chiamati anche **nodi**) di una **lista concatenata** sono disposti secondo un ordine lineare e tutti i nodi sono dello stesso tipo.
- A differenza degli array in cui l'ordine è mantenuto dagli indici dell'array, in una lista, l'ordine è assicurato dal fatto che in ciascun elemento (nodo) è mantenuta l'informazione sull'elemento successivo.
- A differenza di un array, distinti nodi di una lista non occupano locazioni di memoria contigue e la lunghezza di una lista (e, cioè, il suo numero di nodi) può aumentare o diminuire nel corso dell'esecuzione (**struttura dati dinamica**) aggiungendo o rimuovendo nodi.

Introduzione alle liste concatenate (3/4)

Una lista concatenata è caratterizzata dai seguenti aspetti:

- I nodi sono disposti secondo un ordine lineare e sono dello stesso tipo.
- Ciascun nodo è identificabile in modo univoco (nel linguaggio C, un nodo è identificato dal suo indirizzo di memoria iniziale).
- Ad ogni nodo sono associate **due tipologie di informazione**: la prima riguarda i dati veri e propri e la seconda riguarda l'informazione che identifica il nodo successivo (**informazione di connessione o linking**). Nel linguaggio C, tale informazione di linking viene mantenuta tramite un puntatore al nodo successivo.

Introduzione alle liste concatenate (4/4)

- L'ultimo nodo deve essere riconoscibile: e, cioè, l'informazione di linking deve avere un valore che indichi che non c'è più nulla dopo.
- Viene mantenuta l'informazione su chi è il primo nodo nella lista.



Array verso liste concatenate (1/2)

Vantaggi array:

- Accesso diretto agli elementi tramite l'indice di posizione: l'operazione di accesso ad un elemento all'interno dell'array, dato il suo indice, è indipendente dal valore dell'indice di posizione.

Svantaggi array:

- Le operazioni di inserimento e rimozione di elementi quando l'ordine degli elementi deve essere preservato è dispendiosa: ad esempio, assumendo di voler inserire un nuovo elemento alla posizione di indice 50 di un array di lunghezza 200 dove solo i primi 100 elementi hanno valori significativi è necessario spostare in avanti di una posizione gli elementi dalla posizione 50 alla posizione 100.
- La lunghezza di un array non può essere modificata nel corso dell'esecuzione (**struttura dati non dinamica**).

Array verso liste concatenate (2/2)

Vantaggi liste:

- Le operazioni di inserimento e rimozione di un elemento (nodo) sono efficienti dal momento che sono effettuate in maniera ‘*locale*’, cioè, operando solo sugli elementi vicini alla posizione in cui vogliamo inserire un nuovo elemento o da cui vogliamo eliminare un elemento.
- Le liste concatenate sono **dinamiche**, pertanto la lunghezza di una lista può aumentare o diminuire in fase di esecuzione, in base alle necessità.

Svantaggi liste:

- L’accesso agli elementi o nodi di una lista è **sequenziale**. Si può accedere ad un elemento di posizione **i** solo dopo aver acceduto a tutti gli **i-1** elementi precedenti.

Liste concatenate in C (1/3)

- Nel linguaggio C, le liste concatenate sono implementate utilizzando strutture autoreferenziali e l'allocazione dinamica della memoria.
- I nodi di una lista di un certo tipo sono rappresentati tramite variabili di una struttura autoreferenziale avente un campo puntatore (**campo di connessione o link**) allo stesso tipo di struttura.
- Ciascun nodo è identificato dal suo indirizzo iniziale di memoria e il **campo di connessione** punta al nodo successivo nella lista se il nodo non è l'ultimo nella lista e punta a NULL altrimenti.
- L'ultimo nodo della lista è caratterizzato dal fatto che il suo campo di connessione contiene il valore NULL.
- L'informazione sul primo elemento della lista è mantenuta tramite una variabile puntatore al primo elemento della lista. Se la lista è vuota tale puntatore contiene il valore NULL.

Liste concatenate in C (2/3)

- Per illustrare l'implementazione in C delle operazioni fondamentali sulle liste concatenate, utilizzeremo per semplicità una struttura autoreferenziale avente un solo campo dati di tipo **int**. Tale campo verrà utilizzato anche come *campo chiave* per illustrare le operazioni su liste ordinate per valori crescenti del campo chiave.
- Come puntatore al primo nodo della lista utilizzeremo una variabile puntatore al tipo Nodo di nome **pPrimo** definita, ad esempio, come variabile locale automatica nel corpo del main.

```
struct Nodo
{
    //Campo rappresentante i
    // dati della struttura.
    //Utilizzato anche come
    //campo chiave
    int dati;

    //Campo di connessione:
    // puntatore al nodo successivo
    struct Nodo * pSucc;
};
```

Liste concatenate in C (3/3)

Operazioni fondamentali:

- Creazione di un nuovo nodo tramite allocazione dinamica della memoria.
- **Operazioni di inserimento**: inserimento in testa alla lista, inserimento in coda alla lista, e inserimento in mezzo.
- Stampa di una lista.
- **Operazioni di rimozione (o cancellazione)**: rimozione dell'intera lista dalla memoria e cancellazione di un elemento avente una certa chiave.
- **Ricerca lineare**: accesso ad un elemento avente una certa chiave.

Creazione di un nuovo nodo

- Implementiamo una funzione che prende un dato di tipo **int** in ingresso, alloca dinamicamente tramite **malloc** una struttura **Nodo**, inizializza la struttura con il campo **dati** inizializzato al valore dato in ingresso e il campo **pSucc** inizializzato a **NULL**. La funzione restituisce un puntatore alla struttura allocata se l'allocazione dinamica è possibile e restituisce **NULL** altrimenti.

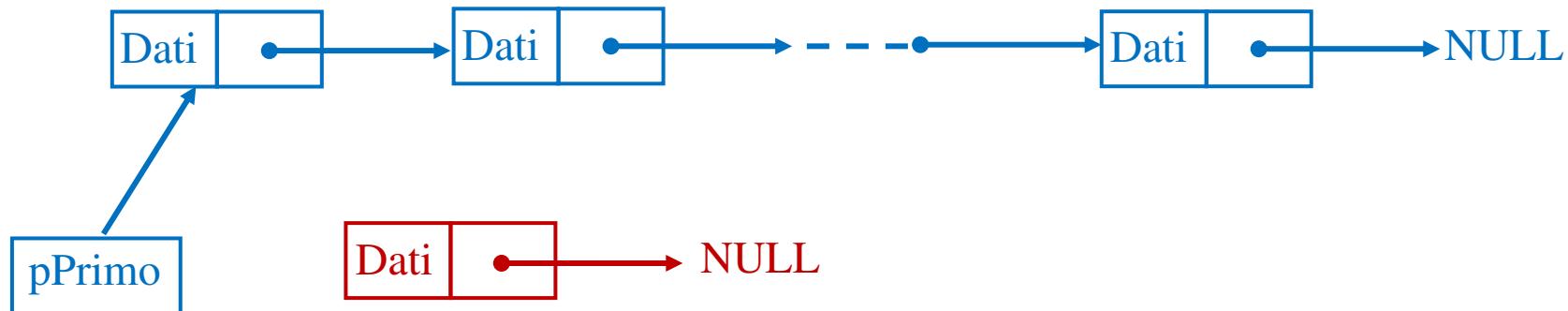
```
struct Nodo * CreaNodo(int dati)
{
    struct Nodo * pNuovoNodo =
        (struct Nodo *) malloc(sizeof(struct Nodo));
    if(pNuovoNodo != NULL)
    {
        pNuovoNodo->dati = dati;
        pNuovoNodo->pSucc = NULL;
    }

    return pNuovoNodo;
}
```

Inserimento in testa di un nuovo nodo (1/3)

- Implementiamo una funzione **InserisciInTesta** che prende in input un puntatore al primo nodo di una lista e un dato di tipo **int** in ingresso, crea un nuovo nodo tramite la funzione **CreaNodo** e inserisce tale nuovo elemento come *primo* nodo della lista. La funzione resituisce un puntatore al primo nodo della lista aggiornata se la creazione del nuovo nodo è possibile e **NULL** altrimenti.

Prima dell'inserimento assumendo che la lista non sia vuota.



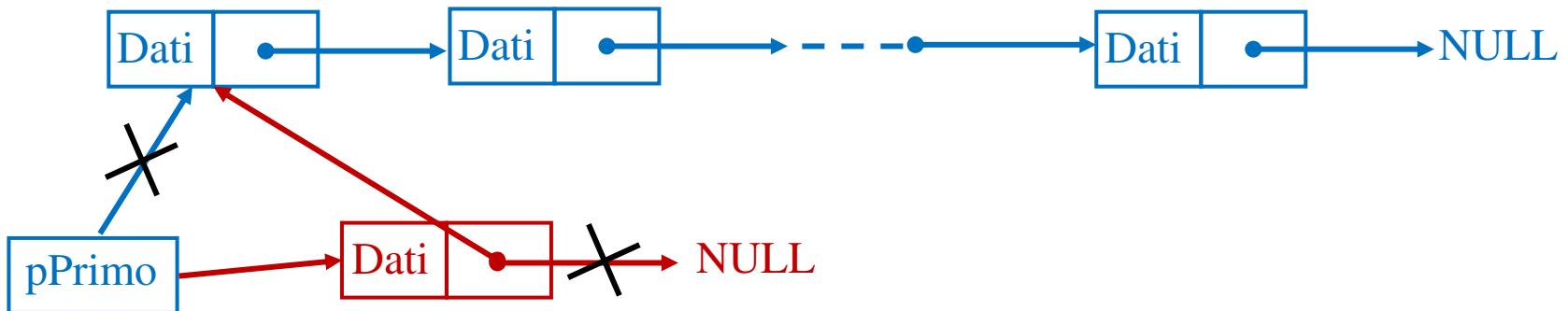
Inserimento in testa di un nuovo nodo (2/3)

```
//pNodo punta al primo elemento della Lista.  
// Se La Lista è vuota pNodo punta a NULL  
struct Nodo * InserisciInTesta(struct Nodo * pNodo,int dati)  
{  
    struct Nodo * pNuovoNodo = CreaNodo(dati);  
    if(pNuovoNodo != NULL)  
        pNuovoNodo ->pSucc = pNodo;  
  
    return pNuovoNodo;  
}
```

Inserimento in testa di un nuovo nodo (3/3)

```
void main()
{
    struct Punto * pPrimo = NULL;
    ...
    struct Nodo * p = InserisciInTesta(pPrimo, 5);
    if(p != NULL)
        pPrimo = p;
}
```

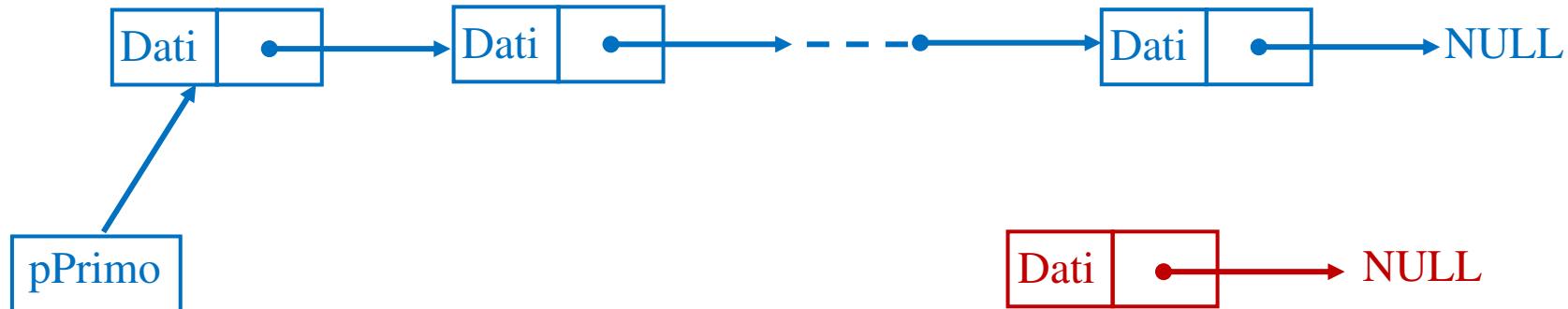
Dopo l'inserimento.



Inserimento in coda di un nuovo nodo (1/3)

- Implementiamo una funzione **InserisciInCoda** che prende in input un puntatore al primo nodo di una lista e un dato di tipo **int** in ingresso, crea un nuovo nodo tramite la funzione **CreaNodo** e inserisce tale nuovo elemento come *ultimo* nodo della lista. La funzione resituisce un puntatore al primo nodo della lista aggiornata se la creazione del nuovo nodo è possibile e **NULL** altrimenti.

Prima dell'inserimento assumendo che la lista non sia vuota.



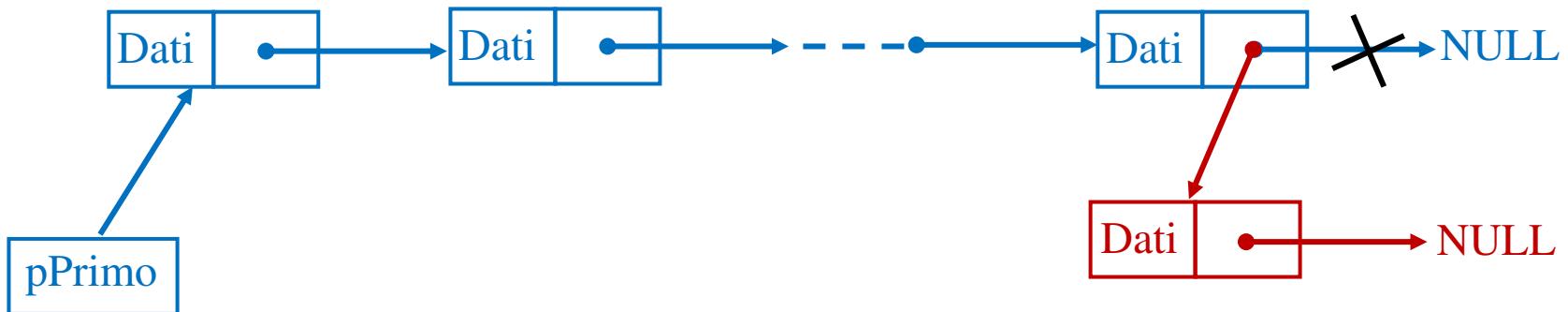
Inserimento in coda di un nuovo nodo (2/3)

```
//pNodo punta al primo elemento della lista.  
// Se la lista è vuota pNodo punta a NULL  
struct Nodo * InserisciInCoda(struct Nodo * pNodo,int dati)  
{  
    struct Nodo * pNuovoNodo = CreaNodo(dati);  
  
    //Se la lista non è vuota e l'allocazione  
    // del nuovo nodo avviene con successo  
    if(pNuovoNodo != NULL && pNodo != NULL)  
    {  
        struct Nodo * pCurr = pNodo;  
        //Scorri tutta la lista  
        while(pCurr->pSucc != NULL)  
            pCurr = pCurr->pSucc;  
  
        //Ora pCurr punta all'ultimo nodo  
        //della lista  
        pCurr ->pSucc = pNuovoNodo;  
        return pNodo;  
    }  
    else  
        return pNuovoNodo;  
}
```

Inserimento in coda di un nuovo nodo (3/3)

```
void main()
{
    struct Punto * pPrimo = NULL;
    ...
    struct Nodo * p = InserisciInCoda(pPrimo, 10);
    if(p != NULL)
        pPrimo = p;
}
```

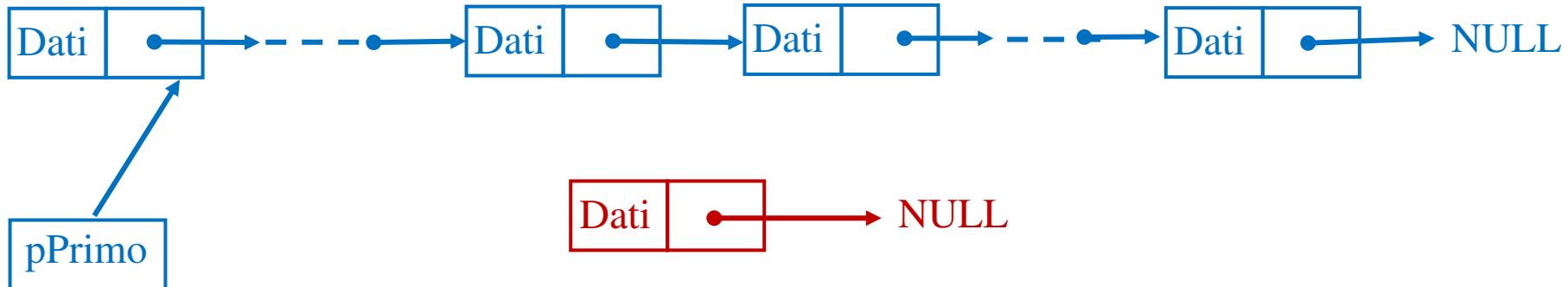
Dopo l'inserimento assumendo che la lista non sia vuota.



Inserimento “in mezzo” di un nuovo nodo (1/4)

- Assumiamo che la nostra lista sia ordinata per valori crescenti del campo dati (utilizzato in questo caso come **campo chiave**).
- Definiamo una funzione **InserisciInOrdine** che prende in input un puntatore al primo nodo della lista e un dato di tipo **int** in ingresso, crea un nuovo nodo tramite la funzione **CreaNodo** e inserisce tale nuovo elemento nella lista preservando l’ordine crescente dei campi chiave.
- La funzione resituisce un puntatore al primo nodo della lista aggiornata se la creazione del nuovo nodo è possibile e NULL altrimenti.

Prima dell’inserimento assumendo che la lista non sia vuota.



Inserimento “in mezzo” di un nuovo nodo (2/4)

- Dopo avere creato un nuovo nodo con il campo dati (utilizzato anche come campo chiave) al valore d’input, la funzione **InserisciInOrdine** procede come segue:
 - se la lista è vuota termina e restituisce il puntatore al nuovo elemento allocato.
 - Altrimenti, utilizza due variabili puntatore ausiliarie **pPrec** e **pCurr** inizializzate rispettivamente a **NULL** e all’indirizzo del primo elemento della lista. Tramite un’iterazione scorre la lista aggiornando ad ogni passo d’iterazione il puntatore **pPrec** a **pCurr** e **pCurr** al nodo successivo nella lista fintanto **pCurr** non è **NULL** e il campo chiave di **pCurr** è minore o uguale al campo chiave del nuovo elemento.
 - All’uscita dal ciclo sono possibili due situazioni: o **pCurr** è **NULL** (il nuovo elemento è il più grande) o **pCurr** è il primo nodo con la chiave più grande del nuovo nodo. Nel primo caso utilizzando la variabile **pPrec**, il nuovo nodo viene inserito alla fine della lista. Nel secondo caso, il nuovo nodo viene inserito tra il nodo puntato da **pPrec** ed il nodo puntato da **pCurr**.

Inserimento “in mezzo” di un nuovo nodo (3/4)

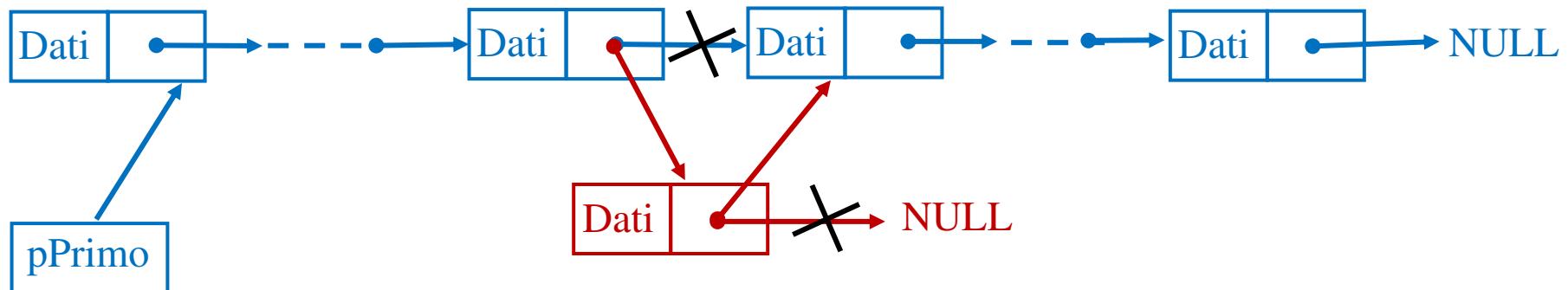
```
//pNodo punta al primo elemento della lista. Se la lista è vuota pNodo punta a NULL
struct Nodo * InserisciInOrdine(struct Nodo * pNodo,int dati)
{
    struct Nodo * pNuovoNodo = CreaNodo(dati);
    //Se la lista non è vuota e l'allocazione
    // del nuovo nodo avviene con successo
    if(pNuovoNodo != NULL && pNodo != NULL)
    {
        struct Nodo * pPrec = NULL;
        struct Nodo * pCurr = pNodo;
        while(pCurr != NULL && pCurr->dati<= dati)
        {
            pPrec= pCurr;
            pCurr = pCurr->pSucc;
        }
        pNuovoNodo->pSucc = pCurr;

        if(pNodo == pCurr)
            return pNuovoNodo;
        else
        {
            pPrec->pSucc = pNuovoNodo;
            return pNodo;
        }
    }
    else
        return pNuovoNodo;
}
```

Inserimento “in mezzo” di un nuovo nodo (4/4)

```
void main()
{
    struct Punto * pPrimo = NULL;
    ...
    struct Nodo * p = InserisciInOrdine(pPrimo, 10);
    if(p != NULL)
        pPrimo = p;
}
```

Dopo l'inserimento.



Stampa di una lista

```
//pNodo punta al primo elemento della lista.  
// Se la Lista è vuota pNodo punta a NULL  
void StampaLista(struct Nodo * pNodo)  
{  
    while(pNodo != NULL)  
    {  
        printf("Valore del nodo = %d\n",pNodo->dati);  
        pNodo = pNodo->pSucc;  
    }  
}
```

Rimozione dell'intera lista

```
//pNodo punta al primo elemento della lista.  
// Se la lista è vuota pNodo punta a NULL  
void RimuoviLista(struct Nodo * pNodo)  
{  
    struct Nodo * pTemp;  
    while(pNodo != NULL)  
    {  
        pTemp=pNodo;  
        pNodo = pNodo->pSucc;  
        free(pTemp);  
    }  
}  
void main()  
{  
    struct Punto * pPrimo = NULL;  
    //...  
    RimuoviLista(pPrimo);  
    //Aggiorna pPrimo a NULL  
    pPrimo=NULL;  
}
```

Rimozione di un elemento con un data chiave (1/2)

- Implementiamo una funzione **TrovaElimina** che prende in input un puntatore al primo nodo di una lista e un dato di tipo **int** in ingresso, e cerca il primo nodo nella lista avente una data chiave e nel caso esista lo rimuove dalla lista.
- La funzione resituisce un puntatore al primo nodo della lista aggiornata se un nodo con la data chiave è trovato e **NULL** altrimenti. Si noti che nel secondo caso la lista non viene modificata.
- L'implementazione della funzione è simile a quella della funzione **InserisciInOrdine**.

Rimozione di un elemento con un data chiave (2/2)

```
//pNodo punta al primo elemento della lista.
struct Nodo * TrovaElimina(struct Nodo * pNodo, int dati)
{
    if(pNodo != NULL)
    {
        struct Nodo * pPrec = NULL;
        struct Nodo * pCurr = pNodo;
        while(pCurr != NULL && pCurr->dati != dati)
        {
            pPrec= pCurr;
            pCurr = pCurr->pSucc;
        }

        if(pCurr != NULL)
        {
            if(pNodo == pCurr)
                pNodo = pCurr->pSucc;
            else
                pPrec->pSucc = pCurr->pSucc;

            free(pCurr);
            return pNodo;
        }
    }
    return NULL;
}
```

Ricerca di un elemento con un data chiave

- Implementiamo una funzione **Trova** che prende in input un puntatore al primo nodo di una lista e un dato di tipo **int** in ingresso, e cerca il primo nodo nella lista avente una data chiave. Se tale nodo esiste, restituisce un puntatore a tale nodo, altrimenti restituisce **NULL**.

```
//pNodo punta al primo elemento della lista.  
struct Nodo * TrovaElimina(struct Nodo * pNodo, int dati)  
{  
    while(pNodo != NULL && pNode->dati != dati)  
        pNode = pNode -> pSucc;  
  
    return pNode;  
}
```

Lezione 18

Antonio Origlia
a.a. 2022/2023

Esercizi su Matrici

1. Scrivere un programma che svolga i seguenti tre punti usando tre funzioni distinte:
 - crei un matrice di interi dinamicamente con memoria **non-contigua**,
 - la riempia con valori assegnati dall'utente,
 - la stampi a video.
2. Come nell'esercizio precedente, ma utilizzando allocazione dinamica con **memoria contigua** per il punto 1 e usando l'algebra dei puntatori per il secondo e terzo punto.

Esercizi su array di stringhe

3. Si scriva un programma in C che acquisisca da tastiera un testo libero, composto da più righe (max 1000). Utilizzare la funzione di libreria **gets** per l'acquisizione da tastiera di una singola riga di caratteri. L'inserimento termina quando l'utente inserirà una riga uguale a FINE. Al termine dell'acquisizione del testo, il programma dovrà stampare le seguenti statistiche:
 - Il numero totale di righe inserite;
 - Il numero totale di caratteri inseriti;
 - Il numero totale di caratteri alfanumerici inseriti;
 - Il numero totale di parole inserite.

Suggerimento: allocare un array di puntatori a caratteri di lunghezza 1000, dove ogni elemento inizializzato dell'array punta al primo carattere di una stringa rappresentante una delle righe inserite da tastiera.

Esercizi su liste concatenate

I seguenti esercizi considerano liste i cui nodi hanno un solo campo dati rappresentato da un intero.

4. Scrivere un programma che definisca le seguenti due funzioni:

- una funzione che prende un intero $N > 0$ in input e crea una lista di N nodi dove i dati associati ai nodi sono forniti da tastiera. La funzione dovrà restituire un puntatore al primo nodo della lista.
- Una funzione che prende in input un puntatore al primo nodo di una lista ed un intero k , rimuove dalla lista tutti i nodi (rilasciando anche la memoria associata) il cui campo dati è minore o uguale a k , e restituisce un puntatore al primo nodo della lista aggiornata.

Testare le due funzioni nel main utilizzando una funzione ausiliaria per la stampa di una lista (la funzione di stampa prende in input un puntatore al primo elemento della lista).

Esercizi su liste concatenate

5. Scrivere una funzione che prenda in input un puntatore al primo nodo di una lista ed un intero k , riordini la lista in modo tale che tutti i nodi con campo dati minore o uguale a k precedano i nodi con campo dati maggiore di k , e restituisca un puntatore al primo nodo della lista aggiornata.
Testare la funzione nel main utilizzando la funzione di allocazione lista del primo punto dell'esercizio 4 e una funzione per la stampa di una lista.
6. Scrivere una funzione che prenda in input un intero $N > 0$ e crei una lista di N nodi ordinata per valori crescenti del campo dati dove i dati associati ai nodi sono forniti da tastiera.
Testare la funzione nel main utilizzando una funzione per la stampa di una lista.

Lezione 20: Esercizi

Laura Bozzelli
a.a. 2020/2021

Esercizi su Array e Matrici (1/4)

Testare nel main le funzioni da implementare nei seguenti esercizi utilizzando funzioni ausiliarie per l'allocazione dinamica di array e matrici e l'inizializzazione con valori da -100 a 100 scelti in modo random.

1. Scrivere una funzione **elaboraArray** che prende in input una matrice quadrata **M** di interi $N \times N$ dove N è dispari, e restituisce un vettore **V** (array monodimensionale) di interi contenente gli elementi situati nelle 4 sottomatrici quadrate (lette per righe) che si ottengono da **M** escludendo la riga e la colonna centrale (si veda l'esempio sottostante).

M =

3	19	13	22	7
15	0	24	2	16
4	21	9	14	23
17	1	25	8	11
6	20	12	18	5

elaboraArray (M) restituisce
V = [3,19,15,0,22,7,2,16,17,1,6,20,8,11,18,5].

Esercizi su Array e Matrici (2/4)

2. Implementare una funzione **verifica** che riceve in input una matrice di interi **M** e un intero **k** positivo e restituisce **true** (valore 1) se **M** contiene tutti gli interi da 1 a **k**, altrimenti restituisce **false** (valore 0).

M =

3	19	13	22	7
15	0	24	2	16
4	21	9	14	23
17	1	25	8	11
6	20	12	18	5

Per la matrice a sinistra e **k=6**, la funzione **verifica** restituisce **true** dal momento che tutti gli interi da 1 a 6 occorrono come elementi della matrice.

Esercizi su Array e Matrici (3/4)

3. Implementare una funzione **elaboraMatrice** che riceve in input una matrice quadrata di interi **M** e restituisce una matrice **A** ottenuta da **M** eliminando gli elementi che appartengono alla diagonale principale.

M =

3	19	13	22	7
15	0	24	2	16
4	21	9	14	23
17	1	25	8	11
6	20	12	18	5

elaboraMatrice (M) restituisce A =

19	13	22	7
15	24	2	16
4	21	14	23
17	1	25	11
6	20	12	18

Esercizi su Array e Matrici (4/4)

4. Implementare una funzione **stampaSottoMatrici** che riceve in input una matrice di interi **M** e stampa a video tutte le sottomatrici quadrate di **M** la cui somma di elementi è maggiore o uguale a zero.
5. Implementare una funzione che prenda in input due matrici quadrate delle stesse dimensioni e restituisca il prodotto righe per colonne delle due matrici.

Esercizi su liste concatenate

Testare nel main le funzioni da implementare nei seguenti esercizi utilizzando funzioni per l'inserimento di dati da tastiera e la stampa di liste concatenate.

6. Scrivere una funzione che prenda in input una lista concatenata di interi e restituisca una nuova lista contenente tutti e solo gli elementi della prima lista che sono numeri primi.
7. Scrivere una funzione che prenda in input due liste di interi ordinate per valori crescenti e costruisca e restituisca una nuova lista di interi ottenuta mediante la “fusione” delle prime due.
8. Scrivere una funzione che prenda in input una lista di interi e la riordini per valori crescenti utilizzando in modo ricorsivo l'algoritmo di ordinamento **MergeSort**.

Lezione 20

Antonio Origlia
a.a. 2022/2023

Sommario - Lezione 20: Gestione file di testo e file binari

- Gestione dei file in C.
- Gestione dei flussi.
- Flussi di testo e flussi binari.
- Funzioni di apertura e chiusura di flussi.
- Lettura e scrittura di file di testo.
- Lettura e scrittura di file binari.
- Funzioni per l'accesso casuale.

Gestione dei file in C

- La memorizzazione dei dati nelle variabili di un programma è *temporanea*: tali dati vano *perduti* al termine dell'esecuzione del programma.
- I **file** sono “contenitori di dati” che vengono usati per la memorizzazione persistente dei dati. I file sono memorizzati su dispositivi di memoria secondaria, come dischi fissi, unità flash e DVD.
- Nel linguaggio C, la creazione, aggiornamento, ed elaborazione di file di dati avviene tramite le funzioni della libreria standard di Input/Output (I/O) i cui prototipi sono dichiarati nel file header **stdio.h**.
- I file sono gestiti dal Sistema Operativo. La realizzazione delle funzioni standard di I/O del linguaggio C tiene conto delle funzionalità del sistema operativo ospite.

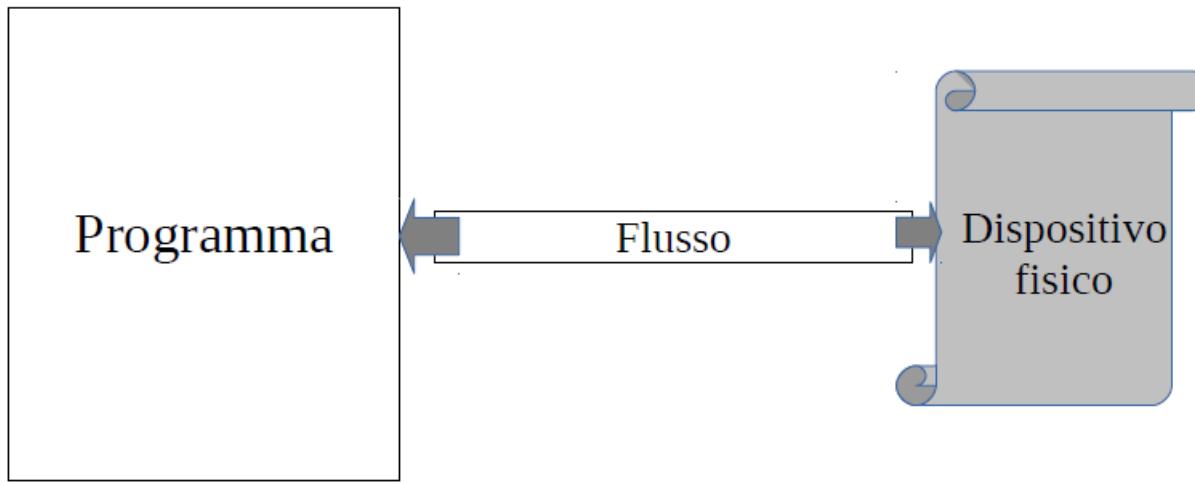
Funzioni di libreria Input/Output (1/2)

Le funzioni di libreria I/O consentono di effettuare operazioni di lettura/scrittura su dispositivi di Input/Ouput in modo indipendente dalle caratteristiche proprie di tali dispositivi.

- Una stessa funzione per operazioni di input può essere utilizzata, ad esempio, sia per leggere un valore dalla tastiera sia per leggere un valore da un dispositivo di memoria di massa tramite un file in esso memorizzato.
- Similmente, una funzione per operazioni di output può essere utilizzata sia per la visualizzazione sullo schermo sia per scrivere su un disco o una stampante.
- Tale astrazione indipendente dal dispositivo effettivo è realizzata tramite un'*interfaccia*, chiamata **flusso** (o **stream**), che consente lo scambio di dati tra il programma ed il dispositivo.

Funzioni di libreria Input/Output (2/2)

Le funzioni di libreria I/O leggono/scrivono dati (sequenze di byte) a/da un dispositivo fisico attraverso i **flussi (stream)**. Uno stream rappresenta il canale di comunicazione tra il programma ed il dispositivo.



- La proprietà fondamentale di tale gestione dell’I/O è che tutti i flussi si comportano nella stessa maniera e, quindi, possono essere gestiti nella stessa maniera indipendentemente dal tipo di dispositivo fisico (stampante, file presente su una memoria di massa, schermo, ecc).

Tipologie di flusso

Esistono due tipologie di flusso o stream:

- **Flussi di testo:** i dati vengono gestiti usando la loro rappresentazione in forma di sequenza di caratteri. Tali sequenze di caratteri sono organizzate in linee. Ogni linea termina con il carattere speciale di “newline” cioè la sequenza di escape ‘\n’.
- **Flussi binari:** i dati sono gestiti esattamente nel modo in cui sono rappresentati in memoria.

Ad esempio, in un flusso di testo, un dato di tipo **int** viene specificato tramite una sequenza di caratteri la cui lunghezza dipende dallo specifico dato (l'intero 125 viene rappresentato con un numero di caratteri diverso da 12). In un flusso binario, differenti dati di tipo **int** vengono rappresentati sempre con lo stesso numero di byte pari a **sizeof(int)**.

Gestione dei flussi

- **Apertura di un flusso:** per associare un flusso ad un dispositivo fisico (ad esempio, un file su disco fisso) è necessaria un'operazione di apertura (creazione dello stream). L'operazione di apertura compie le azioni preliminari necessarie affinchè il dispositivo possa essere acceduto (in lettura o in scrittura).
- **Accesso al flusso:** una volta associato un flusso ad un dispositivo fisico è possibile scambiare dati tra il dispositivo ed il programma.
- **Chiusura del flusso:** una volta terminate le operazioni di lettura/scrittura sul dispositivo, è necessaria un'operazione di chiusura per eliminare l'associazione tra il flusso ed il dispositivo.

Struttura FILE

Le funzioni di I/O utilizzano il tipo struct **FILE** (definito in **stdio.h**) per gestire internamente uno stream e consentire al programmatore il riferimento ad uno specifico stream.

- La funzione **fopen** della libreria di I/O utilizzata per associare un flusso ad un dispositivo fisico, alloca dinamicamente una struttura **FILE** e restituisce un puntatore a tale struttura.
- Una struttura **FILE** ha campi per tenere traccia, in particolare, delle seguenti informazioni:
 - Modalità di accesso al dispositivo (lettura o scrittura).
 - Posizione corrente sul dispositivo (indicante il prossimo byte o carattere da leggere o scrivere sul dispositivo).
 - Un indicatore di **end-of-file** che indica la posizione associata ad un byte utilizzato come marcatore finale dei dati associati al dispositivo.

Apertura di uno stream (1/2)

L'associazione di un flusso con un dispositivo fisico avviene tramite la funzione **fopen** (prototipo definito in **stdio.h**).

Prototipo:

FILE * fopen (char * name, const char * mode):

- **name**: stringa di caratteri indicante il nome del dispositivo fisico per il quale creare lo stream. Per file su disco, il parametro deve essere il nome relativo o il nome assoluto del file nel **file system** (ad esempio “*Desktop\Prova.txt*”).
- **mode**: stringa di caratteri indicante la modalità di accesso.

Apertura di uno stream (2/2)

FILE * fopen (char * name, const char * mode):

- In caso di successo (è possibile creare l'associazione), la funzione alloca dinamicamente una struttura FILE e restituisce un puntatore a tale struttura.
- In caso di errore (non si hanno gli opportuni diritti di accesso per creare l'associazione o il dispositivo è incompatibile con la modalità di accesso indicata), la funzione restituisce il puntatore NULL.

NOTA: prima di accedere ad un dispositivo è necessario assicurarsi che la chiamata alla funzione **fopen** sia stata eseguita con successo, cioè che non abbia restituito NULL.

Apertura in sola lettura

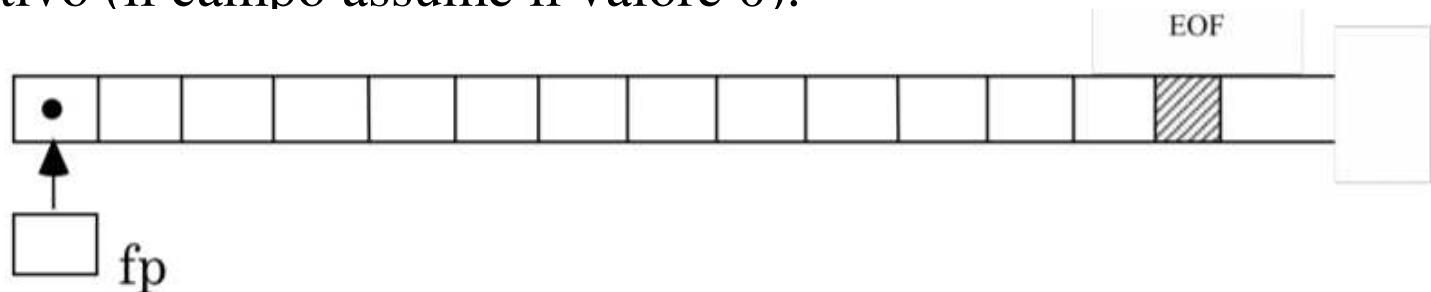
Chiamata per stream di testo:

FILE * fp = fopen (<nome dispositivo>, "r"):

Chiamata per stream binari:

FILE * fp = fopen (<nome dispositivo>, "rb"):

Il dispositivo viene acceduto solo per operazioni di lettura. La funzione restituisce NULL se il dispositivo (ad esempio, un file) non esiste. In caso di successo, per la struttura FILE creata, il campo indicatore di posizione all'interno del dispositivo indica l'inizio del dispositivo (il campo assume il valore 0).



Apertura in sola scrittura

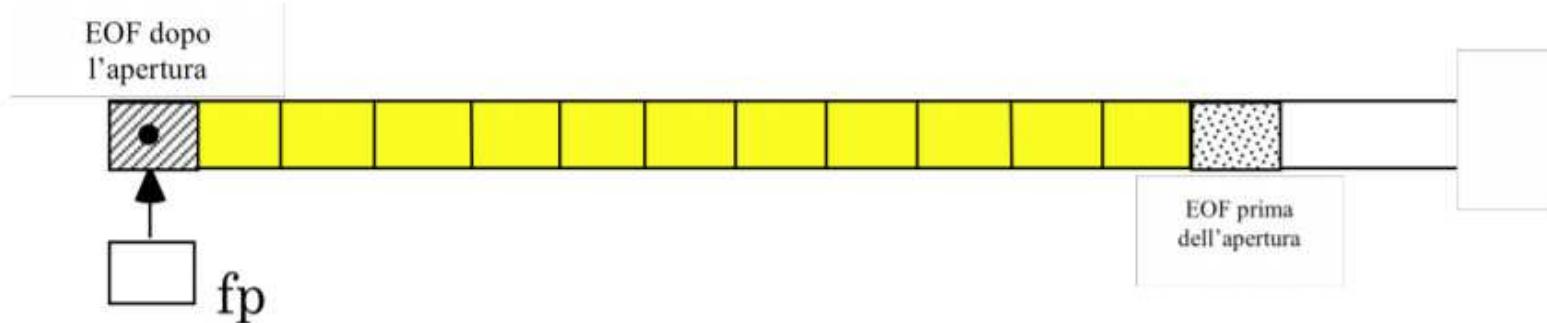
Chiamata per stream di testo:

FILE * fp = fopen (<nome dispositivo>, "w");

Chiamata per stream binari:

FILE * fp = fopen (<nome dispositivo>, "wb");

Il dispositivo viene acceduto solo per operazioni di scrittura. Per file di dati (in formato testo o binario), se il file già esiste, il contenuto corrente viene perso (sovrascritto).



Apertura in sola scrittura con aggiunta

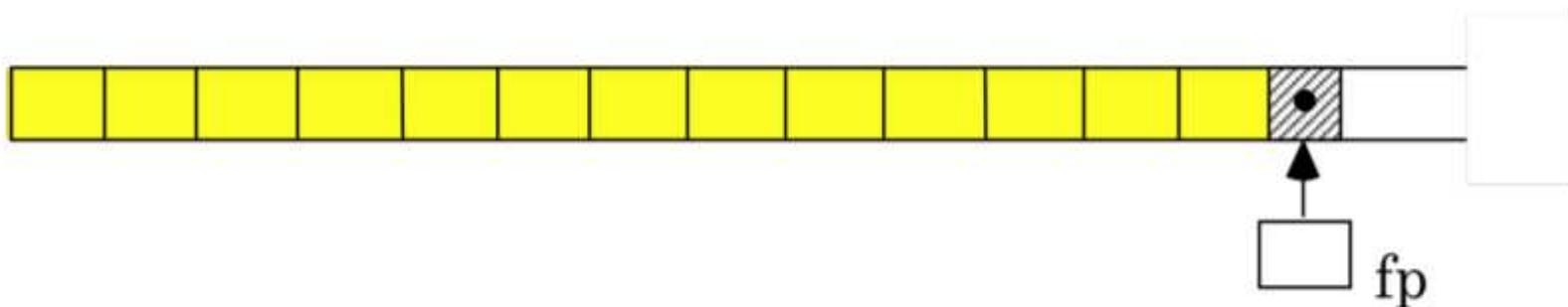
Chiamata per stream di testo:

FILE * fp = fopen (<nome dispositivo>, "a");

Chiamata per stream binari:

FILE * fp = fopen (<nome dispositivo>, "ab");

Il dispositivo viene acceduto solo per operazioni di scrittura. Per file di dati (in formato testo o binario), se il file già esiste, il contenuto corrente non viene perso, e l'indicatore di posizione si porta inizialmente alla fine del file.



Altre modalità di apertura per file

Per stream di testo:

- "**r+**": apre un file di testo già esistente per l'aggiornamento (lettura e scrittura).
- "**w+**": crea un file di testo per l'aggiornamento (lettura e scrittura). Se il file già esiste, il contenuto corrente viene perso.
- "**a+**": crea un file di testo già esistente per l'aggiornamento (lettura e scrittura). Se il file già esiste, il contenuto corrente non viene perso e l'indicatore di posizione si porta inizialmente alla fine del contenuto corrente.

Per stream binari:

- "**rb+**"
- "**wb+**"
- "**ab+**"

Tabella delle modalità di apertura per file

Modalità	Descrizione
r	Apre un file esistente per la lettura.
w	Crea un file per la scrittura. Se il file esiste già, <i>elimina</i> i contenuti correnti.
a	Apre o crea un file per scrivere alla fine del file – cioè, le operazioni di scrittura aggiungono dati al file.
r+	Apre un file esistente per l'aggiornamento (lettura e scrittura).
w+	Crea un file per l'aggiornamento. Se il file esiste già, <i>elimina</i> i contenuti correnti.
a+	Append: apre o crea un file per l'aggiornamento; tutta la scrittura è effettuata alla fine del file – cioè, le operazioni di scrittura aggiungono dati al file.
rb	Apre un file esistente per la lettura in forma binaria.
wb	Crea un file per la scrittura in forma binaria. Se il file esiste già, elimina i contenuti correnti.
ab	Append: apre o crea un file per la scrittura alla fine del file in forma binaria.
rb+	Apre un file esistente per l'aggiornamento (lettura e scrittura) in forma binaria.
wb+	Crea un file per l'aggiornamento in forma binaria. Se il file esiste già, elimina i contenuti correnti.
ab+	Append: apre o crea un file per l'aggiornamento in forma binaria; la scrittura è effettuata alla fine del file.

Flussi (stream) standard

Tre **stream di testo** vengono automaticamente aperti quando inizia l'esecuzione del programma:

- lo **standard input** (riceve input da tastiera): aperto in sola lettura.
- lo **standard output** (stampa output su schermo): aperto in sola scrittura.
- lo **standard error** (stampa messaggi di errore su schermo): aperto in sola scrittura.

Lo standard input, lo standard output, e lo standard error vengono manipolati usando le variabili globali puntatore di tipo FILE * **stdin**, **stdout**, e **stderr** definite nel file header **stdio.h**.

Chiusura di un flusso (1/2)

Al termine di una sessione di accesso ad un dispositivo, lo stream associato e creato tramite la funzione **fopen** deve essere chiuso tramite la funzione **fclose**.

Prototipo di fclose.

```
int fclose (FILE * fp);
```

- **fp** : deve puntare alla struttura **FILE** allocata e restituita dalla chiamata a **fopen** che ha creato lo stream che si vuole chiudere.
- Restituisce come risultato un intero:
 - Se l'operazione di chiusura viene eseguita con successo, il valore restituito è 0;
 - altrimenti, il valore restituito corrisponde a quello associato alla costante simbolica **EOF** definita nel file header **stdio.h**.

Chiusura di un flusso (2/2)

- Se uno stream creato con **fopen** non viene esplicitamente chiuso tramite una chiamata alla funzione **fclose**, il sistema operativo chiuderà automaticamente lo stream al termine dell'esecuzione del programma.
- Comunque la chiusura di uno stream può liberare risorse richieste da altri utenti o programmi. Pertanto, è importante chiudere esplicitamente uno stream quando non è più necessario accedere al relativo dispositivo, invece di aspettare che sia il sistema operativo a chiuderlo al termine dell'esecuzione del programma.

Controllare l'indicatore di end-of-file

In operazioni di lettura da uno stream, per verificare se è stato raggiunto il marcatore finale **end-of-file** dei dati associati al dispositivo, si utilizza la funzione **feof**.

Prototipo di feof.

```
int feof (FILE * fp);
```

- Restituisce un valore maggiore di zero (valore Booleano true) nel caso in cui l'indicatore di posizione si è portato in corrispondenza del marcatore finale **end-of-file**, e 0 altrimenti (valore Booleano false).

Funzioni di scrittura su stream di testo (1/3)

Funzione fprintf

int fprintf (FILE * fp, char * stringa_di_controllo,....);

consente di scrivere testo formattato su uno stream di testo aperto in modalità di scrittura o aggiornamento. Rappresenta una generalizzazione della funzione **printf** per stream di testo associati ad arbitrari dispositivi d'output (in particolare, file di testo). In caso di successo, la funzione restituisce il numero di caratteri scritti nello stream e l'indicatore di posizione si porta in corrispondenza del carattere successivo all'ultimo carattere scritto. Altrimenti, la funzione restituisce **EOF**.

Esempio:

```
int num = 3;
printf("Il numero e\' %d\n",num);
//L'istruzione precedente è equivalente a
fprintf(stdout,"Il numero e\' %d\n",num);
```

Funzioni di scrittura su stream di testo (2/3)

Funzione fputc

```
int fputc (char c, FILE * fp);
```

consente di scrivere il carattere **c** (convertito ad **unsigned char**) su uno stream di testo aperto in modalità di scrittura o aggiornamento. In caso di successo, la funzione restituisce il carattere scritto nello stream e l'indicatore di posizione si porta in corrispondenza del carattere successivo. Altrimenti, la funzione restituisce **EOF**.

Esempio:

```
char c = 'A';
//Scrive il carattere c sullo standard output
fputc(c,stdout);
```

Funzioni di scrittura su stream di testo (3/3)

Funzione fputs

```
int fputs (char * s, FILE * fp);
```

consente di scrivere una stringa di caratteri su uno stream di testo aperto in modalità di scrittura o aggiornamento. In caso di successo, la funzione restituisce un valore positivo e l'indicatore di posizione si porta in corrispondenza del carattere successivo all'ultimo carattere scritto. Altrimenti, la funzione restituisce **EOF**.

Esempio:

```
char s[] = "Ciao mondo!";
//Scrive la stringa s sullo standard output
fputs(s,stdout);
```

Funzioni di lettura da stream di testo (1/3)

Funzione fscanf

```
int fscanf (FILE * fp, char * stringa_di_controllo,.....);
```

consente di leggere testo in modo formattato da uno stream di testo aperto in modalità di lettura o aggiornamento. Rappresenta una generalizzazione della funzione **scanf** per stream di testo associati ad arbitrari dispositivi d'input (in particolare, file di testo). In caso di successo, la funzione restituisce il numero di caratteri letti dallo stream e l'indicatore di posizione si porta in corrispondenza del carattere successivo all'ultimo carattere letto. Altrimenti, la funzione restituisce **EOF**.

Esempio:

```
int val;  
scanf("%d",&val);  
//L'istruzione precedente è equivalente a  
fscanf(stdin,"%d",&val);
```

Funzioni di lettura da stream di testo (2/3)

Funzione fgetc

```
int fgetc (FILE * fp);
```

consente di leggere il carattere correntemente puntato dall'indicatore di posizione da uno stream di testo aperto in modalità di lettura o aggiornamento. In caso di successo (non si è raggiunto il marcitore **end-of-file** di fine dati), la funzione restituisce il carattere letto e l'indicatore di posizione si porta in corrispondenza del carattere successivo. Altrimenti, la funzione restituisce **EOF**.

Esempio:

```
//Legge il carattere corrente dallo standard d'input
char c = fgetc(stdin);
printf("%c",c);
```

Funzioni di lettura da stream di testo (3/3)

Funzione fgets

```
char * fgets (char * s, int n, FILE * fp);
```

consente di leggere dalla posizione corrente di uno stream di testo (aperto in modalità di lettura o aggiornamento) una sequenza di al più **n-1** caratteri e memorizzarla nell'array di caratteri puntato da **s** insieme al carattere '\0' di terminazione stringa. In caso di successo, la funzione restituisce **s**: il numero **k** di caratteri letti può essere inferiore a **n-1** se il **k**-esimo carattere è un newline. In caso di insuccesso (errore o la funzione incontra il marcatore di fine dati), la funzione restituisce NULL.

Esempio:

```
char str[30];
//Inizializza la string str con i primi 30
//caratteri Letti dallo standard d'input
fgets(str,30,stdin);
printf("%s",str);
```

Esempi: stampa di un file di testo

Stampa del contenuto di un file di testo.

```
//Apre in lettura un file di testo
//avente il dato nome e lo stampa a video
void StampaFileTesto(char * nomeFile)
{
    FILE * fp = fopen(nomeFile, "r");
    if(fp != NULL)
    {
        char c;
        while((c = fgetc(fp)) != EOF)
            printf("%c",c);

        fclose(fp);
    }
    else
        printf("Errore nell'apertura del file"
               " di testo %s", nomeFile);
}
```

Esempi: copia di un file di testo

```
//Apre in lettura il file di testo 'sorgente' e lo copia nel file di testo
// 'target'. Se il file di testo 'target' già esiste il suo contenuto viene sovrascritto
void CopiaFileTesto(char * sorgente, char * target)
{
    FILE * fs = fopen(sorgente,"r");
    FILE * ft = fopen(target,"w");
    if(fs == NULL)
    {
        printf("Errore nell'apertura del file"
               " di testo %s", sorgente);
        return;
    }

    if(ft == NULL)
    {
        printf("Errore nell'apertura del file"
               " di testo %s", target);
        return;
    }

    char c;
    while((c = fgetc(fs)) != EOF)
        fprintf(ft,"%c",c);

    fclose(fs);
    fclose(ft);
}
```

Esempi: scrittura e lettura di una matrice (1/3)

Il seguente esempio illustra come scrivere una matrice di interi in un file di testo e come estrarre in lettura la matrice dal file. La struttura del file di testo è come segue:

- La prima linea contiene due interi M e N rappresentanti il numero di righe ed il numero di colonne della matrice, rispettivamente.
- Nelle successive M linee del file di testo sono riportate le M righe della matrice ordinate per valori crescenti dell'indice di riga: sulla seconda linea del file è riportata la prima riga della matrice, sulla terza linea la seconda riga della matrice, e così via.

Ad esempio:

3	4
14	5
67	-21
23	-5
11	10
-1	3
12	18

Esempi: scrittura e lettura di una matrice (2/3)

```
void ScriviMatrice(char * nomeFile, int ** mat,
                    size_t M, size_t N)
{
    FILE * fs = fopen(nomeFile, "w");
    if(fs != NULL)
    {
        fprintf(fs, "%d %d\n", M, N);
        int i, j;
        for(i=0; i<M; i++)
        {
            for(j=0; j<N; j++)
                fprintf(fs, "%d ", mat[i][j]);
            fprintf(fs, "\n"); /* a capo dopo una riga*/
        }

        fclose(fs);
    }
    else
        printf("Errore nell'apertura del file"
               " di testo %s", nomeFile);
}
```

```
void main()
{
    size_t M = 10, N=10;
    int A[M][N];
    int i, j;
    for(i=0; i<M; i++)
    {
        for(j=0; j<N; j++)
            A[i][j] = rand() % 201 - 100;
    }

    int * B[M];
    for(i=0; i<M; i++)
        B[i] = A[i];

    ScriviMatrice("Prova.txt", B, M, N);
}
```

Esempi: scrittura e lettura di una matrice (3/3)

```
int ** LeggiMatrice(char * nomeFile, size_t * pM, size_t * pN)
{
    FILE * fs = fopen(nomeFile,"r");

    if(fs != NULL)
    {
        fscanf(fs, "%d %d",pM, pN);

        int ** mat = (int **) malloc((*pM)*sizeof(int *));
        int i,j;
        for(i=0;i<(*pM);i++)
            mat[i] = (int *) malloc((*pN)*sizeof(int));

        for(i=0;i<(*pM);i++)
        {
            for(j=0;j<(*pN);j++)
                fscanf(fs,"%d ",&mat[i][j]);
        }

        fclose(fs);
        return mat;
    }
    else
    {
        printf("Errore nell'apertura del file"
              " di testo %s", nomeFile);
        return NULL;
    }
}
```

Inizializzazione di stringhe tramite fscanf (1/2)

- Similmente alla funzione **scanf**, è possibile utilizzare lo specificatore di conversione **%s** per inizializzare tramite la funzione **fscanf** un array di caratteri ad una stringa di caratteri dello stream di testo.
- Quando **fscanf** incontra lo specificatore **%s**, legge i caratteri e li memorizza nell'array di caratteri associato finché non incontra uno spazio, una tabulazione, un newline o un indicatore di fine file. A questo punto, la funzione inizializza l'elemento corrente dell'array di input con '**\0**'.

Inizializzazione di stringhe tramite fscanf (2/2)

- È importante assicurarsi che il numero di caratteri processati + il carattere nullo di terminazione non superi la lunghezza del vettore di caratteri (altrimenti si genera un **overflow del buffer**). Ciò può essere garantito utilizzando lo specificatore di conversione per stringhe nel formato %Ns dove N è una costante intera non negativa che indica il numero massimo di caratteri che possono essere letti ed inseriti nell'array di input.

File binari (1/2)

- I file di testo non si prestano bene ad operazioni di modifica che comportano un accesso casuale al file (cambio programmatico dell'indicatore di posizione). Questo perché dati dello stesso tipo sono rappresentati come sequenze di caratteri aventi lunghezze diverse (**dati a lunghezza variabile**).
- Ad esempio, in un file di testo strutturato come una sequenza di *record* contenenti informazioni della stessa tipologia, per modificare un certo record (dal momento che i record sono a lunghezza variabile), è necessario aggiornare tutto il contenuto del file a partire dal record da modificare.

File binari (2/2)

- Nei file binari, dati dello stesso tipo sono rappresentati con lo stesso numero di byte.
- In file binari strutturati come una sequenza di record della stessa tipologia, i record sono a **lunghezza fissa**. È, dunque, possibile modificare un record senza dovere aggiornare l'intero file.
- Inoltre, per record indicizzati da una chiave di ricerca, la posizione esatta di un record all'interno del file può essere calcolata come funzione della chiave del record. Ciò consente l'accesso immediato (**accesso casuale**) a record specifici anche in file di grandi dimensioni.
- Un inconveniente dei file binari è che il numero di byte per rappresentare un tipo primitivo (ad esempio, un intero **int**) dipende dalla macchina soggiacente. Dunque un file binario scritto su una macchina può non essere leggibile su un'altra macchina.

Funzione di scrittura su stream binari

Funzione fwrite

size_t fwrite (void *buffer, size_t n_byte, size_t num, FILE *pf);

- **buffer** rappresenta l'indirizzo iniziale della regione di memoria che contiene i dati da scrivere sullo stream binario riferito da **pf**.
- Il valore di **num** determina il numero di oggetti di ampiezza **n_byte** byte da scrivere sullo stream. Dunque, la regione di memoria allocata puntata da **buffer** deve avere un'ampiezza di almeno **n_byte * num** byte.
- In caso di successo, il valore di ritorno indica il numero di oggetti effettivamente inviati allo stream. Altrimenti, il valore restituito è un numero negativo.

Funzione di lettura da stream binari

Funzione fread

```
size_t fread (void *buffer, size_t n_byte, size_t num, FILE *pf);
```

- **buffer** rappresenta l'indirizzo iniziale della regione di memoria su cui scrivere i dati letti dallo stream binario riferito da **pf**.
- Il valore di **num** determina il numero di oggetti di ampiezza **n_byte** byte da leggere dallo stream. Dunque, la regione di memoria allocata puntata da **buffer** deve avere un'ampiezza di almeno **n_byte * num** byte.
- In caso di successo, il valore di ritorno indica il numero di oggetti effettivamente letti dallo stream. Altrimenti, il valore restituito è un numero negativo.

Esempi: scrittura e lettura di file binari (1/3)

```
typedef struct
{
    unsigned int matricola;
    unsigned int N_esami;
} Studente;

void AggiungiStudente(FILE * pf, Studente st)
{
    fwrite(&st,sizeof(Studente),1,pf);
}

void AggiungiStudenti(FILE * pf, Studente st[],
                      size_t N_studenti)
{
    fwrite(st,sizeof(Studente),N_studenti,pf);
}
```

Esempi: scrittura e lettura di file binari (2/3)

```
void main()
{
    FILE * pf = fopen("studenti.dat","ab");
    if(pf== NULL)
    {
        printf("Il file non puo\` essere aperto \n");
        return;
    }

    int continua;
    do
    {
        printf("Inserisci dati studente!\n");
        Studente st;
        scanf("%u%u",&st.matricola, &st.N_esami);
        AggiungiStudente(pf,st);
        printf("Vuoi continuare (si=1,no=0)?\n");
        scanf("%d", &continua);
    }while(continua);

    close(pf);
    StampaStudenti("studenti.dat");
}
```

Esempi: scrittura e lettura di file binari (3/3)

```
void StampaStudenti(char * nomeFile)
{
    FILE * fs = fopen(nomeFile,"rb");
    if(fs != NULL)
    {
        int result,i=1;
        Studente st;
        while((result = fread(&st,sizeof(Studente),1,fs)) == 1)
        {
            printf(" Studente %d: matricola = %u, numero esami = %u\n",i,
                   st.matricola,st.N_esami);
            i++;
        }
        fclose(fs);
    }
    else
        printf("Errore nell'apertura del file"
               " di testo %s", nomeFile);
}
```

Funzioni per l'accesso casuale (1/2)

Funzione fseek

int fseek(FILE *pf, long offset, int origine);

- Imposta l'indicatore di posizione dello stream associato a **pf** (in termini di numero di byte dall'inizio dello stream). La prossima operazione di I/O sullo stream verrà eseguita dalla nuova posizione impostata.
- La posizione è calcolata aggiungendo **offset** (che può assumere anche valori negativi) a **origine**. Il parametro **origine** può assumere i seguenti valori:
 - **SEEK_SET**: indica l'inizio del file.
 - **SEEK_CUR**: indica la posizione corrente.
 - **SEEK_END**: indica la fine del file.

Funzioni per l'accesso casuale (2/2)

Funzione **ftell**

```
int ftell(FILE *pf);
```

Restituisce il valore corrente dell'indicatore di posizione dello stream associato a **pf** (posizione corrente rispetto all'inizio del file, espressa come numero di byte).

Lezione 22

Antonio Origlia
a.a. 2022/2023

Esercizi su file

1. **Gestione registro esame.** Un registro degli esami è memorizzato in un file di testo con nome “registro.txt” dove sulla prima riga è riportato il numero **N** degli studenti. Ogni altra riga contiene il dato relativo ad un singolo studente, indicando il numero di matricola dello studente (numero intero compreso 1 e 999999), il voto conseguito (numero intero con valore tra 18 e 30). Scrivere un programma che consenta all’utente di scegliere tre opzioni:
 - Opzione ‘media’: il programma deve fornire il voto medio degli studenti promossi (indicato con una sola cifra dopo la virgola).
 - Opzione ‘voto’: in questo caso, l’utente inserisce da tastiera un numero di matricola e il voto conseguito dallo studente. Il programma deve aggiornare il file inserendo i nuovi dati, segnalando però un errore nel caso in cui lo studente abbia già superato l’esame.
 - Uscita dal programma.

Esercizi su file

2. **Gestione registro esame con file binari.** Realizzare la versione dell'esercizio 1 basata su file binari.
3. **Gestione agenda.** Sia dato un file di testo ‘agenda.txt’ contenente gli appuntamenti di una persona ordinati per data e orario. Le informazioni sono così strutturate: ciascun rigo contiene una data (gg/mm) seguita dal numero N di appuntamenti per quella data. Seguono esattamente N coppie ognuna contenente un orario (hh) e una descrizione (in forma di stringa senza spazi di al più 30 caratteri). Scrivere una programma che dati in input due interi (rappresentati la data) e due interi (rappresentanti un orario iniziale e uno finale) restituisca in output la descrizione degli appuntamenti per quella data nell'intervallo di tempo specificato.

<pre>12/05 3 15 Riunione 16 Medico<eoln> 15/06 2 9 Lezione 14 Pranzo<eoln> 18/06 1 15 Riunione<eoln> 18/09 2 9 Lezione 14 Pranzo<eoln><eof></pre>	Input : $(18, 09)$ $(12, 20)$ Output : 14 Pranzo
---	---

Esercizi su file

4. **Gestione parole.** In un file di testo “parole.txt” ogni rigo comincia con un numero che indica quante stringhe (di lunghezza al più 30), separate da uno o più spazi, sono presenti nel rigo stesso.
Realizzare un programma che scriva in un altro file di testo (“output.txt”) solo le righe che contengono più di **k** stringhe dove **k** è dato da tastiera.

Esempio:

Input: k=6

parole.txt:

4 mare palla pallone rete

5 marina sabbia marina mare sabbia

7 fiori petali fiori penne fiori pesci palla

output.txt: fiori petali fiori penne fiori pesci palla

Esercizi su file

5. **Gestione parole variante.** In un file di testo “parole.txt” ogni rigo comincia con un numero che indica quante stringhe (ogni stringa ha lunghezza al più 30), separate da uno o più spazi, sono presenti nel rigo stesso. Realizzare un programma che scriva in un altro file di testo (“output.txt”) solo le righe che non contengono stringhe ripetute.

Esempio:

parole.txt:

4 mare palla pallone rete

5 marina sabbia marina mare sabbia

7 fiori petali fiori penne fiori pesci palla

output.txt: mare palla pallone rete

Esercizi su file

6. **Gestione parole variante difficile.** In un file di testo “parole.txt” ogni rigo contiene un numero variabile di stringhe (ogni stringa ha lunghezza al più 30), separate da uno o più spazi. Realizzare un programma che scriva tre file di testo:
- Il primo (“output1.txt”) contenga tutte e solo le righe di “parole.txt” che non contengono stringhe ripetute.
 - Il secondo (“output2.txt”) contenga le righe di “parole.txt” ma **in ordine inverso**.
 - Il terzo (“output3.txt”) contenga tutte e solo le stringhe che non si ripetono in “parole.txt”.

Lezione 3

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 3: Espressioni e funzioni per l'input/output

- Dichiarazione di variabili semplici.
- Espressioni numeriche.
- Operatore di assegnazione.
- Istruzione di assegnazione.
- Operatori aritmetici.
- Operatori relazionali.
- Operatori logici.
- Conversioni implicite ed esplicite di tipo.
- Funzioni di sistema per l'input/output testuale

Dichiarazione di variabili (tipi semplici)

Una **variabile semplice** è una variabile di tipo semplice (variabile numerica). Una variabile per poter essere utilizzata deve essere prima dichiarata. Una dichiarazione specifica un tipo, un nome simbolico per la variabile, ed, eventualmente, un valore iniziale.

Formato dichiarazione di singola variabile semplice:

<Tipo> <Nome_Variabile>;

- **<Tipo>** denota l'identificatore del tipo di variabile (una parola chiave o due parole chiave per identificare il tipo semplice).
- **<Nome_Variabile>** denota l'identificatore (nome simbolico) scelto per la variabile.

Dichiarazione di variabili (tipi semplici)

Dichiarazione di singola variabile semplice con inizializzazione:

<Tipo> <Nome_Variabile> = <Espressione>;

- **<Espressione>** denota un'*espressione numerica* (definite nel seguito).
- **=** è l'operatore binario di assegnazione che assegna il valore dell'espressione alla variabile (modifica il contenuto della locazione di memoria della variabile).

Esempi:

```
int var1; //dichiara una variabile di nome var1 di tipo int
unsigned long cont; //dichiara una variabile di nome cont di tipo unsigned long
double num = 12.34e-3; //dichiara una variabile di nome num di tipo double e
                      // la inizializza con il valore 12.34e-3
char symbol = 's'; //dichiara una variabile di nome symbol di tipo char e la
                   // inizializza con il codice numerico del carattere 's'.
```

Dichiarazione di variabili (tipi semplici)

Dichiarazione di multiple variabili semplici dello stesso tipo:

<Tipo> <Nome_Variabile₁>, ..., <Nome_Variabile_N>;

Dichiarazione di multiple variabili semplici dello stesso tipo con inizializzazione:

**<Tipo> <Nome_Variabile₁> = <Espressione₁>, ...,
<Nome_Variabile_N> = <Espressione_N>;**

Esempi:

int var1,var2; //dichiara le variabili di nome var1 e var2 di tipo int

long double num =1.3, acc = 12.34e-3; //dichiara le variabili di nome num e acc
// tipo long double e assegna loro i valori 1.3 e
// 12.34e-3 rispettivamente

Dichiarazione di variabili (tipi semplici)

In generale, il valore iniziale di una variabile semplice (e, dunque, numerica) che viene dichiarata senza inizializzazione è:

- Sempre 0 se la variabile è **esterna**, e, cioè, definita al di fuori del corpo di una funzione.
- Valore indefinito (dipende dal compilatore) se la variabile è dichiarata nel corpo di una funzione. Un'eccezione a questa regola è rappresentata dalle **variabile statiche** (definite nel seguito) il cui valore iniziale di default è sempre 0.

Inizializzare sempre le variabili dichiarate nel corpo di una funzione!

Regole per nomi di variabili

Il nome di una variabile può essere un qualunque **identificatore** valido distinto dalle **parole chiave** del C.

Un identificatore è una sequenza di caratteri consistenti in lettere, cifre decimali ed il trattino di underscore _ che NON comincia con una cifra.

Buona pratica: scegliere nomi significativi per le variabili aiuta a rendere un programma auto-documentato e quindi a ridurre la necessità di commenti.

Buona pratica: I nomi di variabili costituiti da più parole possono essere d'aiuto per rendere un programma più leggibile. Es.
elenco_cognomi oppure *elencoCognomi*.

Parole chiave del linguaggio C

Le **parole chiave** del linguaggio C sono parole riservate utilizzate per specificare alcuni costrutti come ad esempio i nomi dei tipi semplici (int, long, double, char, ecc..). Gli identificatori definiti dall’utente (ad esempio, i nomi di variabili) devono essere distinti dalle parole chiave.

Parole chiave

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Parole chiave aggiunte nello standard C99

_Bool _Complex _Imaginary inline restrict

Parole chiave aggiunte nello standard C11

_Alignas _Alignof _Atomic _Generic _Noreturn _Static_assert _Thread_local

Espressioni numeriche

- Un concetto importante dei linguaggi imperativi è quello di **espressione numerica**, la base per eseguire calcoli nei programmi.
- Un'espressione numerica è una combinazione di **operatori predefiniti** del linguaggio binari o unari, di variabili semplici, e costanti numeriche.
- La semantica di un'espressione numerica è espressa in termini di una funzione di valutazione che dato uno stato delle variabili (e, cioè, specifici valori delle variabili) occorrenti in un'espressione *expr*, assegna ad *expr* un unico valore numerico (**valore dell'espressione in un dato stato**).

Sintassi di espressione numerica elementare

Un'espressione numerica elementare può essere:

- un identificatore di una variabile semplice (variabile numerica),
- una costante numerica,
- la *deriferenziazione* di una variabile *puntatore* numerica (lo vedremo nel seguito),
- il componente di un array numerico (lo vedremo nel seguito),
- un campo numerico di una variabile *di tipo aggregato* (lo vedremo nel seguito),
- una chiamata ad una funzione senza argomenti di input che restituisce valori numerici (lo vedremo nel seguito).

Sintassi di espressioni numeriche

Definizione ricorsiva. Un'**espressione numerica** può essere:

- 1.un'**espressione numerica elementare**,
- 2.della forma **<Expr₁> <OpBinario> <Expr₂>**,
- 3.della forma **(< Expr₁>) <OpBinario> (<Expr₂>)**,
- 4.della forma **<OpUnarioPrefisso> <Expr>**,
- 5.della forma **<OpUnarioPrefisso> (<Expr>)**,
- 6.della forma **<Expr> <OpUnarioPostfisso>**,
- 7.della forma **(<Expr>) <OpUnarioPostfisso>**,
- 8.una chiamata ad una funzione che restituisce valori numerici (lo vedremo nel seguito).

dove **< Expr>**, **< Expr₁>** e **<Expr₂>** sono espressioni numeriche,

<OpBinario> è un operatore numerico binario predefinito,

<OpUnarioPrefisso> è un operatore numerico unario-prefisso predefinito,

<OpUnarioPostfisso> è un operatore numerico unario-postfisso predefinito.

Un'*espressione numerica* è ben parentesizzata se è ottenuta applicando le regole sopra ma non le regole 2, 4, e 6.

Valutazione di espressioni numeriche (1/3)

Espressione numerica ben parentesizzata

Data una espressione numerica ben parentesizzata **expr**, per dati valori delle sotto-espressioni elementari occorrenti in **expr**, il valore di **expr** può essere univocamente determinato.

Esempio:

```
int a = 4;  
int b = 1;  
int c = 5;  
int d = ( a + 2) * ( b + c);
```

Il valore iniziale assegnato alla variabile intera **d** è 36.

Valutazione di espressioni numeriche (2/3)

Espressione numerica generica

Data una espressione numerica non parentesizzata **expr**, senza specificare un ordine di valutazione degli operatori, in generale, non è possibile assegnare un unico valore per dati valori delle sotto-espressioni elementari occorrenti in **expr**.

Ad esempio, l'espressione $a + 2 * b + c$ può dar luogo a diverse espressioni ben parentesizzate che forniscono valutazioni diverse:

$(a + 2) * (b + c)$ oppure $(a + (2 * b)) + c$, ecc....

Valutazione di espressioni numeriche (3/3)

Espressione numerica generica

Il C applica un preciso ordine di valutazione ad un'espressione numerica determinato dalle **regole di precedenza** degli operatori (che generalmente sono le stesse di quelle dell'algebra), e per operatori aventi lo stesso livello di precedenza, **regole di associatività da sinistra verso destra o vice versa** (gli operatori in C sono associativi).

Ad esempio, l'espressione $a + 2 * b + c$ viene valutata come l'espressione $a + (2 * b) + c$ ($*$ ha precedenza più alta di $+$).

Le parentesi tonde sono necessarie quando si vuole cambiare l'ordine di valutazione di un'espressione numerica.

Buona pratica: utilizzare le parentesi tonde quando non si ricordano le regole di precedenza degli operatori o le regole di associatività. Ciò serve anche a migliorare la leggibilità del codice.

Operatore di assegnazione per tipi semplici

Sintassi

L'operatore di assegnazione del C, denotato con il simbolo `=`, è un operatore binario utilizzato come tutti gli operatori binari predefiniti in notazione infissa (con il primo operando a sinistra di `=` ed il secondo operando a destra di `=`).

Per tipi numerici (e, cioè, semplici), l'operando a destra può essere una generica espressione numerica, mentre **l'operando a sinistra deve essere un'espressione numerica elementare che ha associata una locazione di memoria**, e cioè,

- o un **identificatore di una variabile semplice** (variabile numerica),
- o la *derenfenziazione* di una variabile *puntatore* numerica (lo vedremo nel seguito),
- o il componente di un array numerico (lo vedremo nel seguito),
- o un campo numerico di una variabile *di tipo aggregato* (lo vedremo nel seguito).

Operatore di assegnazione per tipi semplici

Semantica

L'operatore restituisce il valore dell'operando destro. Inoltre, come **side effect**, il valore dell'operando destro viene assegnato alla locazione di memoria associata all'operando sinistro (**operazione di scrittura della memoria**).

Istruzione di assegnazione per tipi semplici

Istruzione atomica.

Sintassi

<Expr₁> = <Expr₂>;

dove **<Expr₁> = <Expr₂>** è un'espressione numerica e **<Expr₁>** corrisponde all'operando sinistro di = (espressione numerica elementare a cui è associata un'lokazione di memoria).

Semantica

L'esecuzione dell'istruzione corrisponde alla valutazione dell'espressione di assegnazione **<Expr₁> = <Expr₂>** nello stato corrente dell'elaborazione.

Si noti che una dichiarazione di una variabile numerica con inizializzazione corrisponde ad una dichiarazione di variabile senza inizializzazione seguita da un'istruzione di assegnazione.

int b = (a + 2); equivalente a

**int b;
b = a+2;**

Operatori aritmetici binari

Come tutti gli operatori binari hanno notazione infissa.

- **+** : somma.
- **-** : sottrazione.
- ***** : prodotto.
- **/** : divisione. La divisione tra espressioni che assumono valori interi tronca qualsiasi parte frazionaria. Un tentativo di divisione per zero porta ad un errore indefinito che generalmente causa l'interruzione del programma (**crash**).
- **%** : modulo (o operatore di resto). Calcola il resto di una divisione tra interi. Gli operandi devono essere espressioni numeriche che ritornano valori di tipo intero.

Gli operandi di **+, -, *, /** possono essere generiche espressioni numeriche.

Es. **5 / 2** ha come valore 2 e **5 % 2** ha come valore 1.

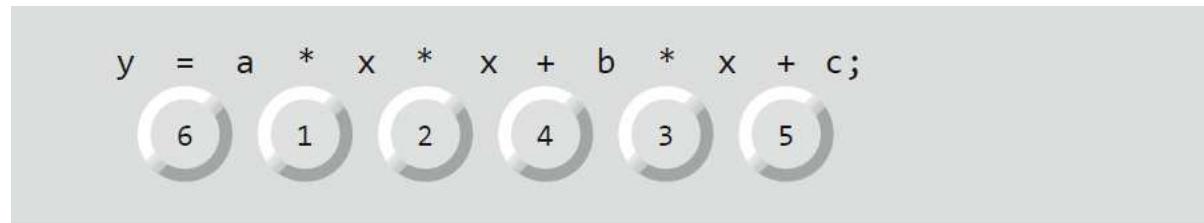
Precedenza e associatività di operatori aritmetici binari

Operatore(i)	Operazione(i)	Ordine di valutazione (precedenza)
()	Parentesi	Valutate per prime. Se le parentesi sono annidate, l'espressione nella coppia <i>più interna</i> è calcolata per prima. Se vi sono diverse coppie di parentesi "allo stesso livello" (cioè non annidate), queste sono calcolate da sinistra a destra.
*	Moltiplicazione	Valutate per seconde. Se ve ne sono diverse, sono calcolate da sinistra a destra.
/	Divisione	
%	Resto	
+	Addizione	Valutate per terze. Se ve ne sono diverse, sono calcolate da sinistra a destra.
-	Sottrazione	
=	Assegnazione	Valutata per ultima.

Esempio di espressione aritmetica (1/2)

Calcolo di un polinomio di secondo grado.

Assumiamo che x, y, a, b, e c siano variabili semplici.



I numeri cerchiati sotto l’istruzione indicano l’ordine in cui vengono eseguite le operazioni in C. Non c’è alcun operatore predefinito per il calcolo della potenza di un numero. Si può utilizzare la funzione di libreria standard **pow**.

Esempio di espressione aritmetica (2/2)

Ordine di calcolo di un polinomio di secondo grado.

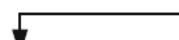
Passo 1. $y = 2 * 5 * 5 + 3 * 5 + 7;$ (Moltiplicazione delle cifre più a sinistra)

2 * 5 è 10



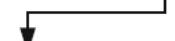
Passo 2. $y = 10 * 5 + 3 * 5 + 7;$ (Moltiplicazione delle cifre più a sinistra)

10 * 5 è 50



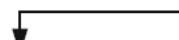
Passo 3. $y = 50 + 3 * 5 + 7;$ (Moltiplicazione prima dell'addizione)

3 * 5 è 15



Passo 4. $y = 50 + 15 + 7;$ (Addizione delle cifre più a sinistra)

50 + 15 è 65



Passo 5. $y = 65 + 7;$ (Ultima addizione)

65 + 7 è 72



Passo 6. $y = 72$ (Ultima operazione – assegnare 72 a y)

Operatori di assegnazione aritmetici composti

Combinano l'operatore di assegnazione (per tipi semplici) con gli operatori aritmetici binari.

Hanno la forma:

<op>=

dove **<op>** denota un operatore aritmetico binario. Dunque, abbiamo gli operatori **+ =**, **- =**, *** =**, **/ =**, **% =**.

L'espressione **<Expr₁> <op>= <Expr₂>** è equivalente a:

<Expr₁> = (<Expr₁> <op> <Expr₂>)

Esempio: se le variabili **x** e **y** hanno valore 3 e 5, rispettivamente, allora il valore dell'espressione **x + = y** è dato da 8. Inoltre, al termine della valutazione il valore 8 è assegnato alla variabile **x**.

Come l'operatore di assegnazione, possono essere utilizzati in istruzioni di assegnazione della forma **<Expr₁> <op>= <Expr₂>;**

Operatori aritmetici unari

- **+** : segno positivo. Notazione prefissa. L'operando può essere una generica espressione numerica. Es. **+ (x / 3.45f)**
- **-** : segno negativo. Notazione prefissa. L'operando può essere una generica espressione numerica. Es. **- (x / 3.45f)**
- **++** : incremento. Aggiunge 1 al valore del suo operando. Può essere utilizzato sia in notazione prefissa che postfissa. L'operando deve essere un'espressione numerica elementare che ha associata una locazione di memoria.
 - Notazione prefissa: l'operatore restituisce come valore il valore dell'operando **dopo** l'incremento. Es. se n vale 5, l'istruzione **x = ++n;** assegna a x il valore 6. **x = ++n;** è equivalente a **n=n+1; x= n;**
 - Notazione postfissa: l'operatore restituisce come valore il valore dell'operando **prima** dell'incremento. Es. se n vale 5, l'istruzione **x = n++;** assegna a x il valore 5. **x = n++;** è equivalente a **x= n; n=n+1;**
 - **--** : decremento. Simile all'incremento ma sottrae 1 al valore del suo operando.

Valori logici

Nel linguaggio C i valori di verità e falsità non sono rappresentati tramite un tipo predefinito.

- Il valore **falso** corrisponde al valore numerico 0.
- Il valore **vero** corrisponde ad un qualsiasi valore numerico diverso da 0.

Operatori relazionali o di confronto

Come tutti gli operatori binari hanno notazione infissa. Gli operandi possono essere generiche espressioni numeriche.

- **>** : maggiore. Restituisce 1 se il valore del primo operando è maggiore del valore del secondo operando, e 0 altrimenti.
- **>=** : maggiore o uguale. Restituisce 1 se il valore del primo operando è maggiore o uguale del valore del secondo operando, e 0 altrimenti.
- **<** : minore. Restituisce 1 se il valore del primo operando è minore del valore del secondo operando, e 0 altrimenti.
- **<=** : minore o uguale. Restituisce 1 se il valore del primo operando è minore o uguale del valore del secondo operando, e 0 altrimenti.
- **==** : uguale. Restituisce 1 se i due operandi hanno lo stesso valore, e 0 altrimenti.
- **!=** : non uguale. Restituisce 1 se due operandi hanno diverso valore, e 0 altrimenti.

Attenzione a non confondere il simbolo = (operatore di assegnazione) con l'operatore di uguaglianza ==.

Operatori logici

Gli operandi possono essere generiche espressioni numeriche.

- `||` : OR (disgiunzione logica). Operatore binario, restituisce 1 se il valore di uno dei due operandi è diverso da zero, e 0 altrimenti.
- `&&` : AND (congiunzione logica). Operatore binario, restituisce 1 se i valori di entrambi gli operandi sono diversi da 0, e 0 altrimenti.
- `!` : NOT (negazione logica). Operatore unario prefisso, restituisce 1 se il valore dell'operando è 0, e 1 altrimenti.

La precedenza di `!` (NOT) è superiore a quella di `&&` (AND) che è a sua volta superiore a quella di `||` (OR).

Gli operatori logici binari `&&` e `||` hanno una precedenza inferiore a quelli relazionali, dove quest'ultimi hanno una precedenza inferiore a quelli aritmetici.

Valutazione cortocircuitata di `&&` e `||`

Le espressioni connesse da `&&` e `||` vengono valutate da sinistra verso destra, e la valutazione si blocca non appena è possibile determinare la verità (valore 1) o la falsità (valore 0) dell'intera espressione.

Esempio: la valutazione dell'espressione

$(x == 1) \&\& (y >= 30)$

si arresterà al termine della valutazione della sottoespressione `x== 1` se il valore di `x` è diverso da 1 (e, cioè, l'intera espressione è falsa, alias assume valore 0), e continuerà se il valore di `x` è 1.

Precedenza e associatività

Precedenza e associatività degli operatori discussi finora.

Gli operatori sono mostrati dall'alto in basso in ordine decrescente di precedenza.

Operatori	Associatività	Tipo
<code>++ (postfisso) -- (postfisso)</code>	da destra a sinistra	postfisso
<code>+ - ! ++ (prefisso) -- (prefisso) (tipo)</code>	da destra a sinistra	unario
<code>* / %</code>	da sinistra a destra	moltiplicativo
<code>+ -</code>	da sinistra a destra	additivo
<code>< <= > >=</code>	da sinistra a destra	relazionale
<code>== !=</code>	da sinistra a destra	di uguaglianza
<code>&&</code>	da sinistra a destra	AND logico
<code> </code>	da sinistra a destra	OR logico
<code>? :</code>	da destra a sinistra	condizionale
<code>= += -= *= /= %=</code>	da destra a sinistra	di assegnazione

Operatori per espressioni condizionali (spiegati in seguito).

Conversione implicita di tipo (1/4)

- Quando utilizzato all'interno di un'espressione numerica, un operatore restituisce un valore di un certo tipo che dipende dai tipi associati ai suoi operandi.
- Per operatori unari, il tipo associato all'operatore in un dato contesto coincide con il tipo del suo operando.
- A differenza di linguaggi fortemente tipizzati come C# o Java, nel linguaggio C un operatore binario può essere applicato ad operandi di tipo diverso.
- Quando un operatore binario ha operandi di tipo diverso, questi vengono convertiti in un tipo comune in accordo ad un insieme ristretto di regole: il valore di uno dei due operandi viene convertito per essere consistente con il tipo del secondo operando.

Conversione implicita di tipo (2/4)

Conversioni per operatori binari che non coinvolgono assegnazione.

Se l'operatore ha operandi di tipo diverso, il valore dell'operando avente tipo "inferiore" viene convertito nel tipo "superiore".

short → int → long → float → double → long double

char → int

<tipo> → unsigned <tipo>

Dove <tipo> denota un tipo intero con segno (short, int, long, char). Per tipi unsigned, la conversione è sicura, e, cioè, avviene senza perdita di informazione.

Conversione implicita di tipo (3/4)

Conversioni per operatori binari che non coinvolgono assegnazione.

Esempio: int x; double y;

Nel calcolo dell'espressione $x+y$,

1. x viene convertito in double,
2. viene effettuata la somma tra valori di tipo double,
3. il risultato è di tipo double.

Le regole di conversione si complicano in presenza di operandi unsigned. Il risultato di un operatore applicato ad operandi con e senza segno dipende dalla macchina, perché legato alle ampiezze dei diversi tipi di interi.

Conversione implicita di tipo (4/4)

Conversioni per operatori di assegnazione.

In questo caso, il valore dell'operando destro viene trasformato nel tipo del valore dell'operando sinistro.

In questo caso, è possibile che un valore di tipo "superiore" venga convertito in un valore di tipo "inferiore" con possibile perdita di informazione ed errori di **overflow**.

- Per interi, i bit più significativi in eccesso vengono rimossi.
- Dai reali ad interi, si ha il troncamento della parte frazionaria.

Conversione esplicita di tipo

Operatore unario di conversione esplicita di tipo.

In un'espressione numerica è possibile forzare specifiche conversioni tramite un operatore unario (**operatore di cast**).

Sintassi

(**<Tipo>**) **<Expr>**

dove **<Tipo>** denota un tipo semplice ed **<Expr>** un'espressione numerica.

Semantica

Converte il valore dell'espressione **<Expr>** in un valore consistente con il tipo specificato **<Tipo>**.

Esempio:

```
float media; int somma = 205; int n=12;  
media = somma/n ; //divisione tra interi  
media = (float)somma/n; //divisione tra reali
```

Istruzione condizionale semplice (1/2)

Le istruzioni eseguibili o eseguono azioni (calcoli, operazioni di input e output) o prendono decisioni. Le istruzioni che prendono decisioni consentono di modificare il flusso dell'esecuzione in dipendenza del soddisfacimento o meno di una certa condizione. La più semplice istruzione di questo tipo è l'**istruzione condizionale semplice**.

Sintassi

```
if(<condizione>)
```

```
{
```

```
    <corpo>
```

```
}
```

dove **<condizione>** è un'espressione numerica generica, e **<corpo>** è una sequenza arbitraria di istruzioni.

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione condizionale con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Istruzione condizionale semplice (2/2)

Sintassi

```
if(<condizione>
{
    <corpo>
}
```

Semantica

- Se l'espressione **<condizione>** è vera (e, cioè, il valore restituito è diverso da 0), il corpo **<corpo>** dell'istruzione viene eseguito. Altrimenti, il corpo non viene eseguito.
- Indipendentemente dal fatto che il corpo venga eseguito o meno, al termine dell'esecuzione dell'istruzione condizionale, l'esecuzione procede con l'istruzione successiva all'istruzione condizionale.

Costante stringa

Nel linguaggio C, una costante stringa è una sequenza arbitraria di caratteri (che può comprendere anche caratteri speciali come le sequenze di escape) racchiusa tra doppie virgolette ". Per includere il carattere ", utilizziamo la sequenza di escape \".

Le sequenze di escape non rappresentano caratteri visibili ma sono usate per la formattazione del testo. Ricordiamo che alcune sequenze di escape fungono da comandi durante interazioni testuali in operazioni di input ed output. Ad esempio, il carattere *new line* consente di posizionare il **cursore (prompt) di lettura/scrittura** nella finestra di interazione all'inizio della riga successiva a quella corrente.

Funzioni di sistema per l'input/output

Introduciamo ora due funzioni della libreria standard del C che consentono di eseguire operazioni di input/output basate su interazione con l'utente di tipo testuale.

- Funzione **printf**: consente di inviare testo formattato in uscita su un dispositivo di output standard (tipicamente lo schermo).
- Funzione **scanf**: consente di ricevere e convertire input testuale fornito dall'utente tramite un dispositivo d'input standard (solitamente la tastiera).

La lettera **f** alla fine dei nomi delle due funzioni sta per formattato.

Funzione printf: sintassi

Sintassi formato ristretto dell’istruzione di chiamata a printf:

printf(<formato>,<arg₁>,....,<arg_N>);

- **<formato>** è una costante stringa chiamata **stringa di controllo del formato**.
- ciascun argomento **<arg_i>** ($1 \leq i \leq N$) alla destra della stringa di controllo può essere o un’espressione numerica oppure una costante stringa.
- $N \geq 0$. Il numero di argomenti d’input alla destra della stringa di controllo è arbitrario e può essere eventualmente nullo.

Funzione printf: semantica (1/2)

`printf(<formato>,<arg1>,....,<argN>);`

Semantica dell'istruzione di chiamata a printf:

- La stringa di formato è costituita da caratteri ordinari (incluse le sequenze di escape) che vengono semplicemente stampati nell'ordine di apparizione sul dispositivo d'output e sotto-sequenze di caratteri che iniziano con il carattere %, chiamate **specifiche di conversione**. Ciascuna specifica di conversione provoca la conversione e la stampa del valore dell'argomento `<argi>` corrispondente.
- Il numero di specifiche di conversione deve corrispondere al numero di argomenti d'input alla destra della stringa di controllo, e ciascuna specifica di conversione deve essere consistente con il tipo dell'associato argomento. In caso contrario, il comportamento della funzione **printf** è indefinito.

Funzione printf: semantica (2/2)

`printf(<formato>,<arg1>,....,<argN>);`

Semantica dell'istruzione di chiamata a printf:

- L'esecuzione dell'istruzione di chiamata a printf comporta la stampa sul dispositivo d'output della sequenza di caratteri ottenuta dalla stringa di controllo sostituendo ogni **specifiche di conversione** con una stringa di caratteri che rappresenta il valore dell'associato argomento `<argi>` consistentemente con la specifica di conversione.
- Per inviare il carattere speciale `%`, si può utilizzare la sequenza `%%`.

Specifiche di conversione di printf

Ogni specifica di conversione inizia con il carattere % e termina con un carattere di conversione, chiamato **specificatore di conversione**. Tra % ed il carattere di conversione possiamo eventualmente trovare, nell'ordine da sinistra a destra:

- Un segno meno – per l'allineamento a sinistra dell'argomento convertito.
- Un numero che specifica l'ampiezza minima del campo. L'argomento convertito viene stampato in un campo di ampiezza almeno pari a quella data. Se necessario, vengono lasciati degli spazi bianchi a sinistra o a destra a seconda dell'allineamento per raggiungere l'ampiezza desiderata.
- Un punto per separare l'ampiezza dal campo della precisione.
- Un numero, la **precisione**, che per argomenti di tipo stringa, specifica il numero di caratteri che devono essere stampati, per argomenti di tipo reale, specifica il numero di cifre dopo il punto decimale, e per argomenti di tipo intero, specifica il numero minimo di cifre di un intero.
- Per un argomento di tipo intero, il carattere ‘h’ se l'intero deve essere stampato come *short*, oppure il carattere ‘l’ (elle) se l'intero deve essere stampato come *long*.

Specificatori di conversione di printf (1/5)

Specificatori di conversione per tipi interi.

Specificatore di conversione	Descrizione
d	Stampa come <i>un intero decimale con segno</i> .
i	Stampa come <i>un intero decimale con segno</i> . [Nota: gli specificatori i e d sono diversi quando sono usati con <code>scanf</code> .]
o	Stampa come <i>un intero ottale senza segno</i> .
u	Stampa come <i>un intero decimale senza segno</i> .
x o X	Stampa come <i>un intero esadecimale senza segno</i> . X fa sì che siano stampate le cifre 0-9 e le lettere <i>maiuscole A-F</i> , e x fa sì che siano stampate le cifre 0-9 e le lettere <i>minuscole a-f</i> .
h, l o ll (lettera “elle”)	Vanno posti <i>prima</i> di uno specificatore di conversione di interi per indicare che viene stampato, rispettivamente, un intero short, un intero long o un intero long long. Questi sono chiamati modificatori di lunghezza .

Specificatori di conversione di printf (2/5)

Esempio specificatori di conversione per tipi interi.

```
int main(void)
{
    printf("%d\n", 455);
    printf("%i\n", 455); // i come d in printf
    printf("%d\n", +455); // non viene stampato il segno piu'
    printf("%d\n", -455); // viene stampato il segno meno
    printf("%hd\n", 32000);
    printf("%ld\n", 2000000000L); // suffisso L per letterale long
    printf("%o\n", 455); // ottale
    printf("%u\n", 455);
    printf("%u\n", -455);
    printf("%x\n", 455); // esadecimale con lettere minuscole
    printf("%X\n", 455); // esadecimale con lettere maiuscole
}
```

Output

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Specificatori di conversione di printf (3/5)

Specificatori di conversione per tipi reali.

Specificatore di conversione	Descrizione
e o E	Stampa un valore in virgola mobile in <i>notazione esponenziale</i> .
f o F	Stampa i valori in virgola mobile nella <i>notazione in virgola fissa</i> (lo specificatore F è supportato nel compilatore Visual C++ di Microsoft in Visual Studio 2015 e versioni successive).
g o G	Stampa un valore in virgola mobile o nel <i>formato f</i> oppure nel formato esponenziale e (o E), in base alla grandezza del valore.
L	Va posto prima dello specificatore di conversione di numeri in virgola mobile per indicare che viene stampato un valore in virgola mobile <i>long double</i> .

Specificatori di conversione di printf (4/5)

Esempio specificatori di conversione per tipi reali.

```
int main(void)
{
    printf("%e\n", 1234567.89);
    printf("%e\n", +1234567.89); // non viene stampato il segno piu'
    printf("%e\n", -1234567.89); // viene stampato il meno
    printf("%E\n", 1234567.89);
    printf("%f\n", 1234567.89); // sei cifre a destra del punto decimale
    printf("%g\n", 1234567.89); // stampa con la lettera minuscola e
    printf("%G\n", 1234567.89); // stampa con la lettera maiuscola E
}
```

Output

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

Specificatori di conversione di printf (5/5)

Specificatori di conversione per altri tipi.

- **c**: tipo carattere.
- **s**: tipo stringa.
- **p**: tipo puntatore (lo vedremo in seguito).

Esempio

```
char character = 'A'; // inizializza un char
printf("%c\n", character);

printf("%s\n", "This is a string");
```

Output

A
This is a string

Funzione scanf: sintassi

La funzione **scanf** è l'analogo, ma per l'input, della funzione **printf**.

Sintassi formato ristretto dell'istruzione di chiamata a scanf:

scanf(<formato>, &<arg₁>, ..., &<arg_N>);

- **<formato>** è una costante stringa chiamata **stringa di controllo del formato**.
- ciascun argomento **<arg_i>** ($1 \leq i \leq N$) alla destra della stringa di controllo può essere un'espressione numerica elementare a cui è associata una locazione di memoria (ad esempio, l'identificatore di una variabile semplice).
- $N \geq 0$. Il numero di argomenti d'input alla destra della stringa di controllo è arbitrario e può essere eventualmente nullo.
- Il simbolo **&** che precede ogni espressione **<arg_i>** ($1 \leq i \leq N$) alla destra della stringa di controllo rappresenta **l'operatore di indirizzamento** (lo vedremo nel seguito). Il suo utilizzo sta a significare che l'argomento d'input viene passato per riferimento (lo vedremo nel seguito).

Funzione scanf: semantica (1/3)

`scanf(<formato>,&<arg1>,....,&<argN>);`

Semantica dell'istruzione di chiamata a scanf:

- La stringa di formato è costituita da caratteri ordinari (incluse le sequenze di escape) e sotto-sequenze di caratteri che iniziano con il carattere %, chiamate **specifiche di conversione**.
- Il numero di specifiche di conversione, ad eccezione delle specifiche di conversione con **il carattere * di soppressione** deve corrispondere al numero di argomenti d'input alla destra della stringa di controllo, e ciascuna specifica di conversione deve essere consistente con il tipo dell'associato argomento. In caso contrario, il comportamento della funzione **scanf** è indefinito.

Funzione scanf: semantica (2/3)

`scanf(<formato>,&<arg1>,....,&<argN>);`

Semantica dell'istruzione di chiamata a scanf:

- La funzione **scanf** legge caratteri dallo standard input, li interpreta in base al contenuto della stringa di controllo e memorizza i risultati nelle locazioni di memoria associate agli argomenti **<arg_i>** alla destra della stringa di controllo.
- Ai caratteri ordinari nella stringa di controllo devono essere associati gli stessi caratteri dallo standard input.
- Ad una specifica di conversione nella stringa di controllo deve essere associato un **campo** e, cioè, una sequenza di caratteri dallo standard d'input priva di caratteri di spaziatura (e cioè spazi, i caratteri new line, return, tab verticale, tab orizzontale, e salto pagine). Il campo deve corrispondere ad un valore consistente con la specifica di conversione. La funzione memorizza tale valore nella locazione di memoria dell'argomento **<arg_i>** associato.

Funzione scanf: semantica (3/3)

`scanf(<formato>,&<arg1>,....,&<argN>);`

Semantica dell'istruzione di chiamata a scanf:

- Un campo dallo standard input si estende fino al primo carattere di spaziatura oppure fino all'ampiezza massima, se specificata nella specifica di conversione corrispondente.
- La funzione **scanf** termina quando esaurisce la sua stringa di controllo oppure quando riscontra un'inconsistenza fra l'input fornito dall'utente e la stringa di controllo.
- La funzione **scanf** ignora tutti i caratteri di spaziatura sia nelle stringa di controllo che nell'input fornito dall'utente.

Specifiche di conversione di scanf

Ogni specifica di conversione inizia con il carattere % e termina con un carattere di conversione, chiamato **specificatore di conversione**. Tra % ed il carattere di conversione possiamo eventualmente trovare, nell'ordine da sinistra a destra:

- Un carattere * di soppressione. Indica che il campo d'input corrispondente viene ignorato. Ad esso, dunque, non è associato nessun argomento alla destra della stringa di controllo.
- Un numero che specifica l'ampiezza massima del campo.
- Per un argomento di tipo intero, il carattere ‘h’ se l'intero deve essere interpretato come *short*, oppure il carattere ‘l’ (elle) se l'intero deve essere interpretato come *long*.
- Per un argomento di tipo reale, il carattere ‘L’ se il numero reale deve essere interpretato come un *long double*.

Specificatori di conversione di scanf

Simili a quelli utilizzati per la funzione printf ma con le seguenti differenze.

- Differenza tra gli specificatori di conversione **d** e **i** per gli interi: **d** legge un intero in notazione decimale con o senza segno, mentre **i** può leggere un intero in notazione decimale, ottale o esadecimale con o senza segno.

Esempio

```
int main()
{
    int num1; //primo numero inserito dall'utente
    int num2; // secondo numero inserito dall'utente

    scanf("%d %d",&num1,&num2);

    if(num1== num2)
    {
        printf("%d e\' usguale a %d",num1,num2);
    }

    if(num1!= num2)
    {
        printf("%d non e\' usguale a %d",num1,num2);
    }
    return 0;
}
```

Lezione 5

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 5: Strutture di controllo (prima parte)

- Pseudocodice e diagrammi di flusso.
- Istruzioni di selezione if e if-else.
- Istruzioni if-else annidate.
- Istruzione di iterazione while.
- Ciclo while controllato da contatore.
- Ciclo while controllato da sentinella.
- Espressioni condizionali.

Algoritmi

Prima di scrivere un programma software per risolvere un problema specifico, è necessaria una comprensione del problema ed un approccio sistematico e pianificato alla sua soluzione.

La risoluzione di un problema implica la definizione di una procedura automatizzabile, **algoritmo**, consistente nell'esecuzione di una serie di azioni in un ordine specifico. La struttura dell'ordine di esecuzione può essere articolata come segue:

- **Ordine sequenziale**: le azioni vengono eseguite nell'ordine in cui sono scritte.
- **Ramificazione condizionale**: un blocco di azioni può essere eseguito o meno dipendendo dal soddisfacimento o meno di una certa condizione.
- **Ripetizione**: un blocco di azioni può essere eseguito ripetutamente finché una certa condizione rimane soddisfatta.

Pseudocodice

- Linguaggio artificiale ed informale che aiuta a sviluppare algoritmi.
- Lo pseudocodice consiste semplicemente in una specifica informale delle azioni da eseguire e del loro flusso di esecuzione.
- Programmi in pseudocodice aiutano a riflettere ed ad acquisire una comprensione profonda del problema da risolvere.
- Programmi in pseudocodice preparati con cura si possono convertire facilmente in corrispondenti programmi in linguaggio C.
- Come pseudocodice utilizzeremo un linguaggio informale con rappresentazione grafica basata sui cosiddetti **diagrammi di flusso**.

Diagrammi di flusso (1/2)

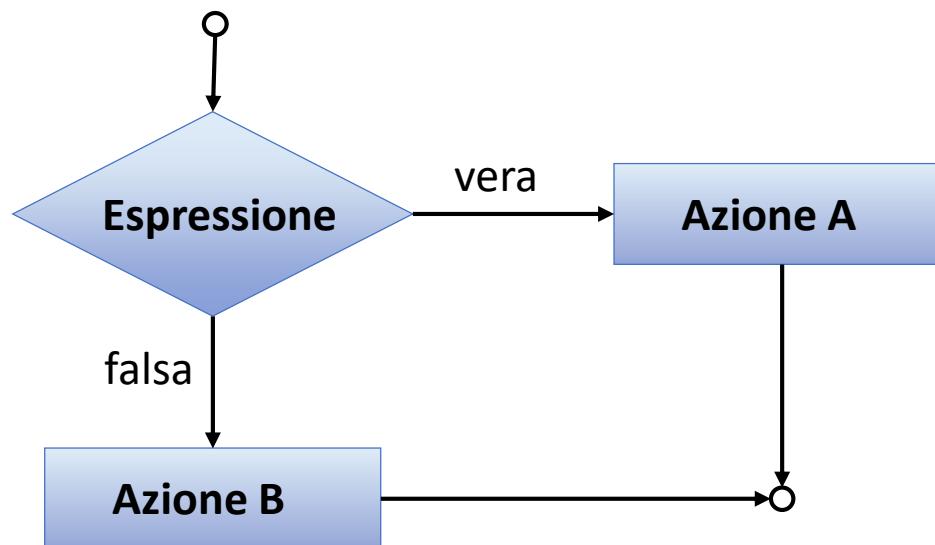
Un diagramma di flusso è una rappresentazione grafica di un algoritmo. Mostrano chiaramente l'ordine di esecuzione delle azioni (**flusso di esecuzione**).

Gli elementi grafici che costituiscono un diagramma di flusso sono:

- **Rettangoli**: per rappresentare azioni. Il testo all'interno del rettangolo è una specifica informale di una certa azione.
- **Rettangoli arrotondati**: per definire il punto di ingresso e di uscita dell'algoritmo.
- **Simboli cerchietto**: per definire il punto di ingresso e di uscita di una porzione dell'algoritmo.
- **Rombi**: chiamato anche **punto di decisione**. Il testo all'interno di un rombo rappresenta un'espressione, come una condizione, che può essere vera o falsa.

Diagrammi di flusso (2/2)

- Gli elementi grafici (rettangoli, rombi, e cerchietti) di un diagramma di flusso sono collegati da frecce chiamate **linee di flusso** che specificano l'ordine di esecuzione delle azioni.
- Un simbolo di decisione (rombo) ha *due* linee di flusso che emergono da esso. Una indica la direzione da prendere quando l'espressione scritta nel simbolo è vera, e l'altra la direzione da prendere quando l'espressione è falsa.



Flusso di esecuzione nel linguaggio C

Nel linguaggio C, le **istruzioni atomiche**, cosiddette perchè non sono decomponibili (corrispondono ad azioni elementari), che compaiono nel corpo di una funzione possono essere classificate in:

- Istruzioni di dichiarazione (ed inizializzazione) di variabili.
- Istruzioni di assegnazione.
- Chiamate di funzioni.

Normalmente, le istruzioni sono eseguite una dopo l'altra nell'ordine in cui sono scritte (**modalità di esecuzione sequenziale**). Il linguaggio C fornisce varie istruzioni non-atomiche, chiamate in generale **istruzioni di controllo**, per specificare un'ordine di esecuzione non sequenziale (**modalità di traferimento del controllo**) :

- **Istruzioni di selezione o condizionali:** per specificare una ramificazione condizionale nel flusso di controllo.
- **Istruzioni di iterazione:** per consentire la ripetizione condizionale di una sequenza di azioni.

Istruzioni di selezione

Consentono di specificare che sequenze di istruzioni debbano essere eseguite o meno in dipendenza del soddisfacimento o meno di una certa condizione (**ramificazione condizionale nel flusso di controllo**).

- Istruzione di selezione semplice **if**: un'alternativa.
- Istruzione di selezione doppia **if-else**: due alternative.
- Istruzione di selezione plurima **switch**: multiple alternative (la vedremo nelle prossime lezioni).

NOTA: sono tutte istruzioni ad un solo ingresso e a una sola uscita.

Istruzione di selezione semplice if (1/3)

Sintassi

```
if(<condizione>)
```

```
{
```

```
    <corpo>
```

```
}
```

dove **<condizione>** è un'espressione numerica generica, e **<corpo>** è una sequenza arbitraria di istruzioni (istruzioni atomiche o istruzioni di controllo).

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione di selezione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Istruzione di selezione semplice if (2/3)

Sintassi semplificata quando il corpo consiste di una sola istruzione

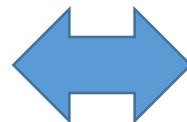
if(<condizione>)

<corpo>

<corpo> deve consistere di un'unica istruzione (atomica o di controllo).

Esempio:

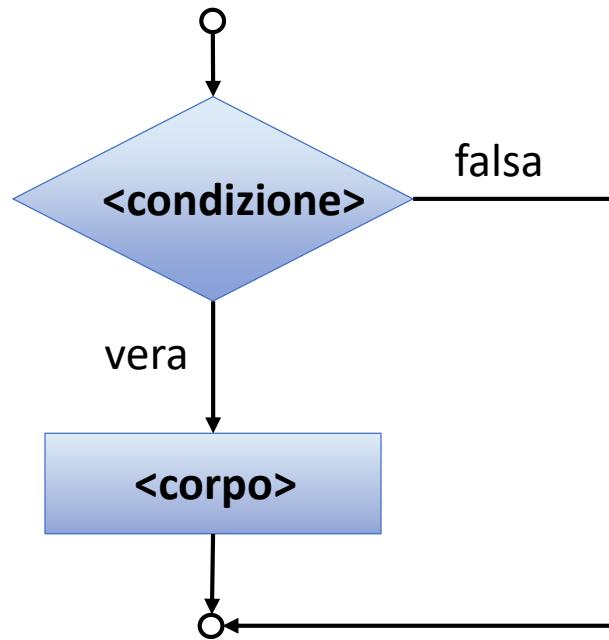
```
if(x>0)
{
    if(z>0)
    {
        z+=x;
    }
}
```



```
if(x>0)
if(z>0)
    z += x;
```

Istruzione di selezione semplice if (3/3)

```
if(<condizione>
{
    <corpo>
}
```



Semantica

- Se l'espressione **<condizione>** è vera (e, cioè, il valore restituito è diverso da 0), il corpo **<corpo>** dell'istruzione viene eseguito. Altrimenti, il corpo non viene eseguito.
- Indipendentemente dal fatto che il corpo venga eseguito o meno, al termine dell'esecuzione dell'istruzione di selezione, l'esecuzione procede con l'istruzione successiva all'istruzione **if**.

Istruzione di selezione doppia if-else (1/3)

Sintassi

```
if(<condizione>
{
    <corpo 1>
}
else
{
    <corpo 2>
}
```

dove **<condizione>** è un'espressione numerica generica, e **<corpo 1>** (*corpo dell'if*) e **<corpo 2>** (*corpo dell'else*) sono sequenze arbitrarie di istruzioni (istruzioni atomiche o istruzioni di controllo).

Buona norma: far rientrare a destra il testo dell'intero corpo **<corpo 1>** (risp., **<corpo 2>**) con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

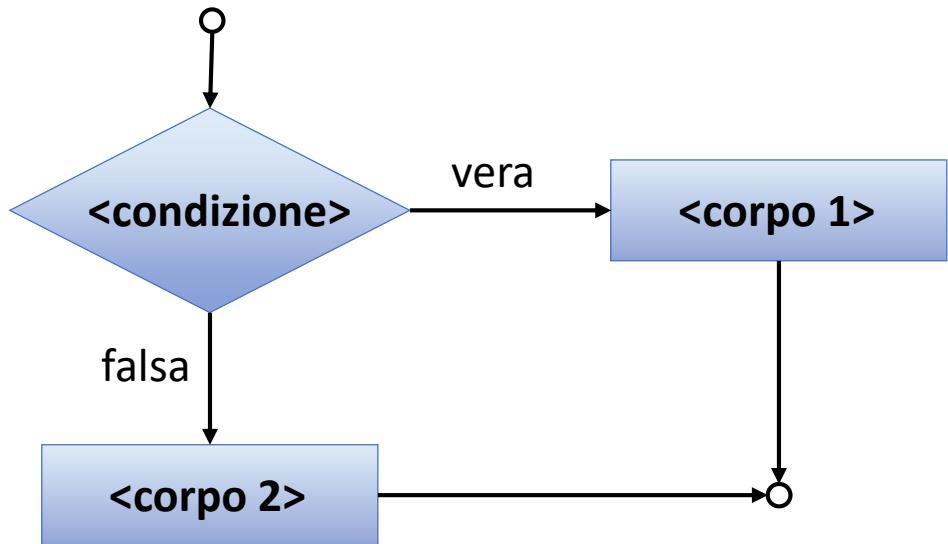
Istruzione di selezione doppia if-else (2/3)

```
if(<condizione>
{
    <corpo 1>
}
else
{
    <corpo 2>
}
```

se **<corpo 1>** consiste di un'unica istruzione è possibile omettere le parentesi graffe che lo delimitano. Similmente per il corpo **<corpo 2>**.

Istruzione di selezione doppia if-else (3/3)

```
if(<condizione>
{
    <corpo 1>
}
else
{
    <corpo 2>
}
```



Semantica

- Se l'espressione **<condizione>** è vera (e, cioè, il valore restituito è diverso da 0), il corpo **<corpo 1>** della *parte if* viene eseguito. Altrimenti, viene eseguito il corpo **<corpo 2>** della *parte else*.
- Al termine dell'esecuzione dell'istruzione di selezione, l'esecuzione procede con l'istruzione successiva all'istruzione **if-else**.

Esempio

Dati due interi positivi (inseriti da tastiera), stabilire se uno è multiplo dell'altro e stampare tale informazione a video.

```
int num1 =0;
int num2=0;

printf("Inserisci due numeri interi positivi\n");
int risultato = scanf("%d %d",&num1,&num2);

if(risultato !=2 || num1 <= 0 || num2 <=0)
{
    printf("I dati inseriti non sono corretti\n");
}
else
{
    if(num1 >= num2 && num1 % num2 == 0)
    {
        printf("%d e' un multiplo di %d \n",num1,num2);
    }
    else
    {
        if(num2 >= num1 && num2 % num1 == 0)
        {
            printf("%d e' un multiplo di %d \n",num2,num1);
        }
        else
        {
            printf("Dei due interi inseriti, nessuno dei due e' multiplo dell'\\'altro");
        }
    }
}
```

Esempio semplificato

Dati due interi positivi (inseriti da tastiera), stabilire se uno è multiplo dell'altro e stampare tale informazione a video.

```
int num1 =0;
int num2=0;

printf("Inserisci due numeri interi positivi\n");
int risultato = scanf("%d %d",&num1,&num2);

if(risultato !=2 || num1 <= 0 || num2 <=0)
    printf("I dati inseriti non sono corretti\n");
else
{
    if(num1 >= num2 && num1 % num2 == 0)
        printf("%d e\' un multiplo di %d \n",num1,num2);
    else
        if(num2 >= num1 && num2 % num1 == 0)
            printf("%d e\' un multiplo di %d \n",num2,num1);
        else
            printf("Dei due interi inseriti, nessuno dei due e\' multiplo dell'\\'altro");
}
```

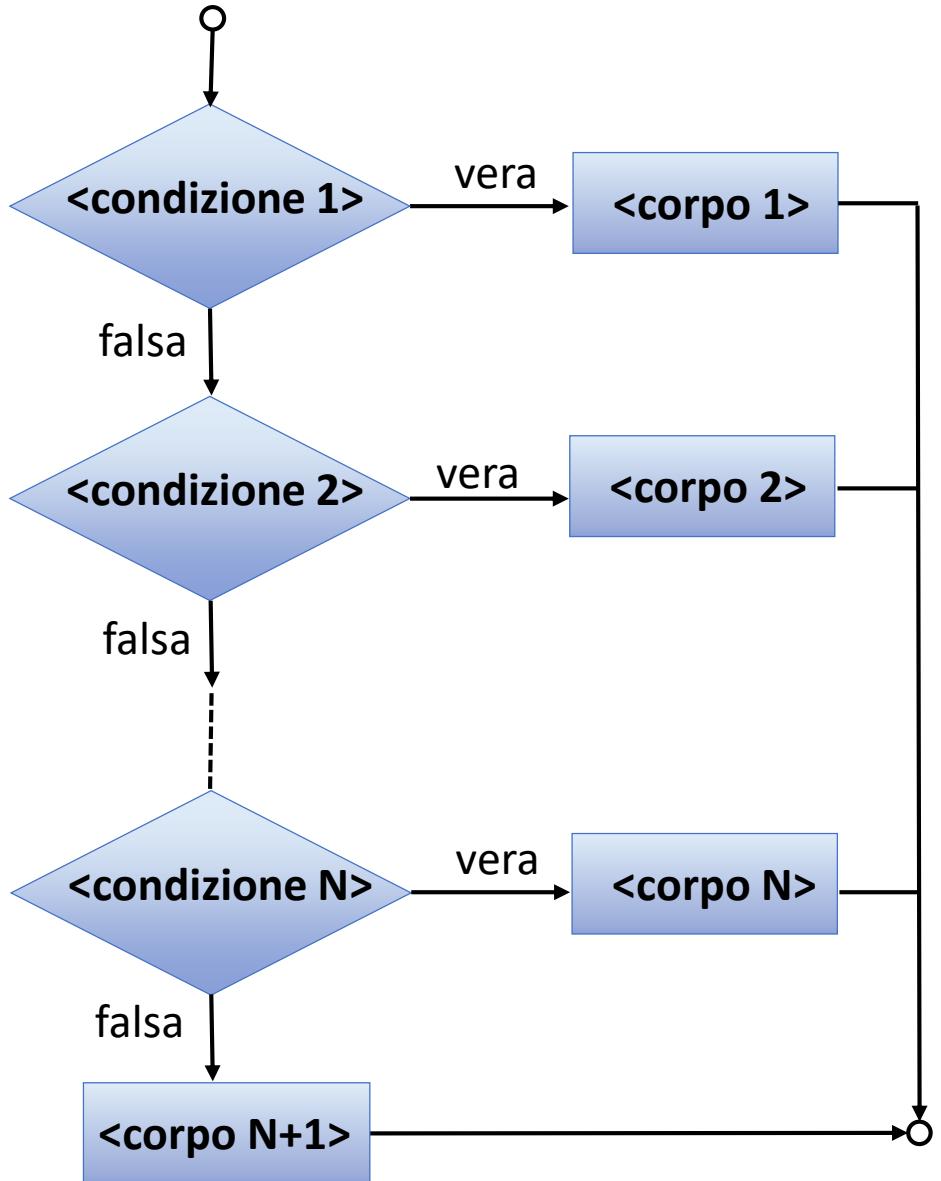
Istruzioni If-else annidate (1/3)

Nell'esempio precedente, il corpo della parte *else* di un'istruzione *if-else* è essa stessa un'istruzione *if-else*. Quest'ultima potrebbe avere come corpo della parte *else* un'altra istruzione *if-else* e così via.

Si parla in questo caso di **istruzione if-else annidata**. Utile quando si effettuano test su casi multipli. Dal momento che i corpi delle varie parti *else* delle istruzioni *if-else* annidate rappresentano singole istruzioni, invece di aggiungere un ulteriore livello di indentazione, è preferibile per chiarezza utilizzare uno stile di scrittura come riportato nella slide seguente.

Istruzioni if-else annidate (2/3)

```
if(<condizione 1>)
{
    <corpo 1>
}
else if(<condizione 2>)
{
    <corpo 2>
}
.....
else if(<condizione N>)
{
    <corpo N>
}
else
{
    <corpo N+1>
}
```



Istruzioni If-else annidate (3/3)

```
if(<condizione 1>)
{
    <corpo 1>
}
else if(<condizione 2>)
{
    <corpo 2>
}
.....
else if(<condizione N>)
{
    <corpo N>
}
else
{
    <corpo N+1>
}
```

Modo più generale per realizzare una scelta tra multiple alternative. Le espressioni **<condizione 1>**,..., **<condizione N>** vengono valutate nell'ordine in cui si presentano; se una di esse risulta vera, il corpo ad essa associato viene eseguito. Altrimenti, viene eseguito il corpo dell'ultima parte *else* (alternativa di default).

NOTA: l'ultima parte *else* può essere omessa.

Esempio ulteriormente semplificato

Dati due interi positivi (inseriti da tastiera), stabilire se uno è multiplo dell'altro e stampare tale informazione a video.

```
int num1 =0;
int num2=0;

printf("Inserisci due numeri interi positivi\n");
int risultato = scanf("%d %d",&num1,&num2);

if(risultato !=2 || num1 <= 0 || num2 <=0)
    printf("I dati inseriti non sono corretti\n");
else if(num1 >= num2 && num1 % num2 == 0)
    printf("%d e\' un multiplo di %d \n",num1,num2);
else if(num2 >= num1 && num2 % num1 == 0)
    printf("%d e\' un multiplo di %d \n",num2,num1);
else
    printf("Dei due interi inseriti, nessuno dei due e\' multiplo dell'\\'altro");
```

Istruzioni di iterazione

Consentono di ripetere l'esecuzione di una sequenza di istruzioni finchè una certa condizione rimane soddisfatta.

- Istruzione di iterazione **while** (detta anche **ciclo while**).
- Istruzione di iterazione **do...while** (la vedremo nelle prossime lezioni).
- Istruzione di iterazione **for** (la vedremo nelle prossime lezioni).

NOTA: sono tutte istruzioni ad un solo ingresso e a una sola uscita.

Istruzione di iterazione while (1/4)

Sintassi

```
while(<condizione>
{
    <corpo>
}
```

dove **<condizione>** è un'espressione numerica generica, e **<corpo>** è una sequenza arbitraria di istruzioni (istruzioni atomiche o istruzioni di controllo).

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione di selezione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Istruzione di iterazione while (2/4)

Sintassi semplificata quando il corpo consiste di una sola istruzione

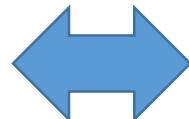
while(<condizione>)

<corpo>

<corpo> deve consistere di un'unica istruzione (atomica o di controllo).

Esempio:

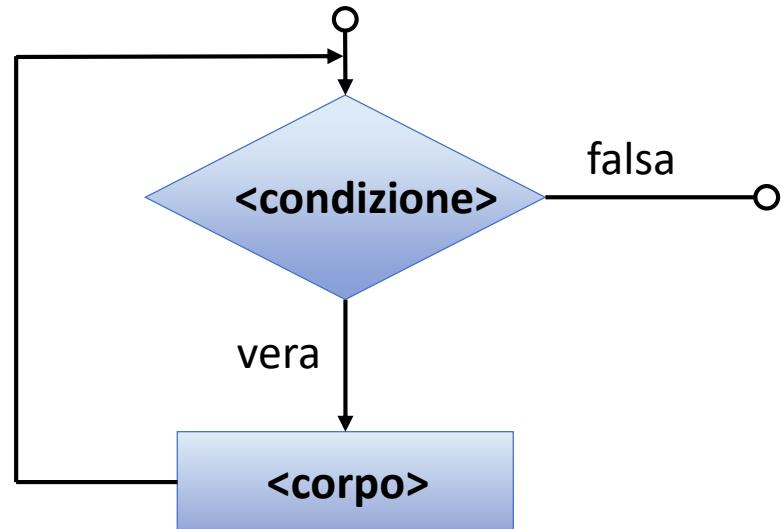
```
while(x>0)
{
    z += x--;
}
```



```
while(x>0)
    z += x--;
```

Istruzione di iterazione while (3/4)

```
while(<condizione>
{
    <corpo>
}
```

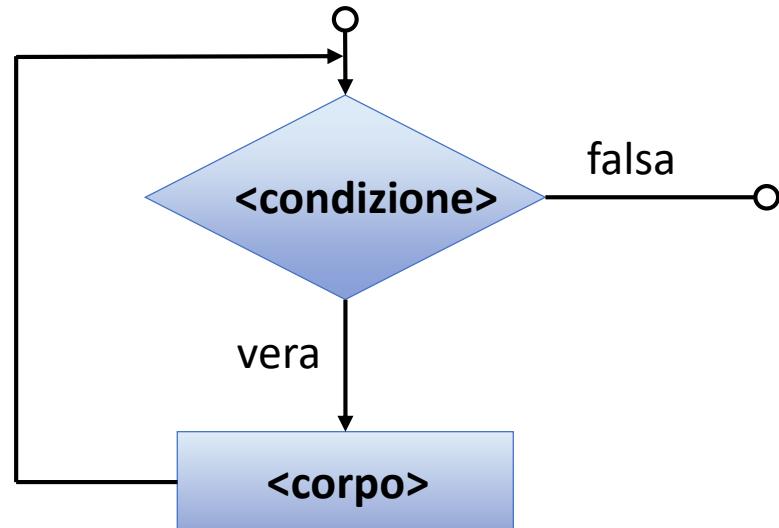


Semantica

- Se l'espressione **<condizione>** è vera (e, cioè, il valore restituito è diverso da 0), il corpo **<corpo>** dell'istruzione viene eseguito. Al termine dell'esecuzione del corpo, l'istruzione *while* viene eseguita nuovamente (nuova iterazione dell'istruzione *while*).
- Se l'espressione **<condizione>** è falsa, l'istruzione *while* termina **(condizione di terminazione)**.
- Al termine dell'esecuzione dell'istruzione *while*, l'esecuzione procede con l'istruzione successiva all'istruzione *while*.

Istruzione di iterazione while (4/4)

```
while(<condizione>)
{
    <corpo>
}
```



Semantica

- Il numero di iterazioni può essere infinito se l'espressione **<condizione>** è sempre vera (**ciclo infinito**).
- L'espressione **<condizione>** conterrà in generale variabili accedute in scrittura nel corpo del ciclo while. Altrimenti, o il corpo non viene mai eseguito (l'espressione **<condizione>** è sempre falsa) oppure si ha un ciclo infinito (l'espressione **<condizione>** è sempre vera).

Ciclo while controllato da contatore (1/3)

- Si utilizza una variabile intera **contatore** inizializzata a 0.
- Si assicura che il numero di iterazioni del *ciclo while* sia pari ad un certo numero N come segue:
 - utilizzando come condizione del ciclo *while*, l'espressione **contatore < N**
 - incrementando di uno la variabile contatore ad ogni iterazione.

```
int contatore = 0;  
while(contatore<N)  
{  
    contatore++;  
    <corpo>  
}
```

Le istruzioni in **<corpo>** devono accedere solo in lettura alla variabile **contatore**.

Ciclo while controllato da contatore (2/3)

Esempio: *dati un intero positivo N (inserito da tastiera), calcolare la somma dei primi N interi e stampare tale somma a video.*

Soluzione del problema:

DATI:

- Variabile intera **num** per memorizzare l'intero N fornito in input.
- Variabile intera **somma** per memorizzare il risultato finale ed i risultati parziali durante il calcolo della somma (**variabile accumulatore**).
- Variabile intera **contatore** per tenere traccia del valore da aggiungere alla variabile **somma** nell'iterazione corrente del ciclo e per assicurare che il numero di iterazioni sia N.

PROCEDIMENTO:

Memorizzare in **num** l'intero N fornito in input

Inizializzare **somma** a 0 e **contatore** a 0

Finchè il valore di **contatore** è minore di **num** fai

 Incrementare di uno il valore di **contatore**

 Aggiungi a **somma** il valore di **contatore**

Stampa il valore di **somma**

Ciclo while controllato da contatore (3/3)

Esempio: dati un intero positivo N (inserito da tastiera), calcolare la somma dei primi N interi e stampare tale somma a video.

```
int num =0;
int somma =0;
int contatore =0;

printf("Inserisci un numero intero positivo\n");
//Prendi in input un numero intero positivo
int res = scanf("%d",&num);

if(res != 1 || num<=0)
    printf("Il dato inserito non e' corretto\n");
else
{
    while(contatore <num)
    {
        contatore++;
        somma += contatore;
    }

    printf("La somma dei primi %d interi e' \n %d \n",num,somma);
}
```

Curiosità

L'algoritmo illustrato per calcolare la somma dei primi N numeri naturali è ottimale?

Risposta: NO

Per ogni numero naturale N, esiste una formula chiusa per calcolare la somma dei primi N interi, e, cioè, una formula algebrica con un numero di operazioni da effettuare indipendente da N. Nel caso particolare, tale formula è un polinomio di secondo grado.

$$\sum_{j=0}^N j = \frac{N(N+1)}{2}$$

Ciclo controllato da sentinella (1/5)

Utilizzato da algoritmi che prendono in input uno o più dati ad ogni iterazione di un ciclo.

Non conoscendo il numero esatto dei dati da prendere in input, deve essere fornito in input un valore speciale, chiamato **valore sentinella**, per indicare la *fine dei dati forniti in input*.

Valore sentinella: deve essere un valore diverso dai possibili valori dei dati in input.

Ciclo controllato da sentinella (2/5)

Schema algoritmo con ciclo controllato da sentinella:

1. Prendi in input il primo dato
2. Finchè il valore in input è diverso dal valore sentinella
 3. Elabora il dato corrente
 4. Prendi in input il prossimo dato.

Esempio: Calcolare la media dei voti di un gruppo di studenti.

L'utente invia in input i voti, uno alla volta, quando richiesto. Ad ogni passo, viene richiesto all'utente di inserire il prossimo dato. Quando l'inserimento dei voti validi è terminato, alla successiva richiesta da parte del programma, l'utente inserisce il valore sentinella.

Ciclo controllato da sentinella (3/5)

Esempio: *Calcolare la media dei voti di un gruppo di studenti.*

Soluzione del problema:

DATI:

- Variabile intera **voto** per memorizzare il dato fornito correntemente in input (eventualmente la sentinella).
- Variabile intera **totale** per memorizzare la somma parziale dei voti correntemente inseriti (**variabile accumulatore**).
- Variabile intera **numVoti** per tenere traccia del numero di voti ricevuti.

Ciclo controllato da sentinella (4/5)

Esempio: *Calcolare la media dei voti di un gruppo di studenti.*

Soluzione del problema:

PROCEDIMENTO:

Inizializzare **totale** a 0 e **numVoti** a 0.

Ricevi in ingresso il primo dato e memorizzalo in **voto**

Finchè il valore di **voto** è diverso dal valore sentinella

 Incrementare di uno il valore di **numVoti**

 Aggiungi a **totale** il valore di **voto**

 Ricevi in ingresso il dato successivo e memorizzalo in **voto**

Se il valore di **numVoti** è maggiore di zero

 Stampa il **totale** diviso il valore di **numVoti**

altrimenti

 Stampa "nessun voto ricevuto"

Ciclo controllato da sentinella (5/5)

Esempio: Calcolare la media dei voti di un gruppo di studenti.

```
int voto = -1;
int totale = 0;
int numVoti = 0;

printf("Inserisci il primo voto\n");
int res = scanf("%d",&voto);

while(res == 1 && voto>=0)
{
    numVoti++;
    totale += voto;
    printf("Inserisci il prossimo voto\n");
    res = scanf("%d",&voto);
}

if(numVoti >0)
    printf("La media dei voti e' %.1f\n", (float)totale/numVoti) ;
else
    printf("Non e' stato inserito alcun voto!\n");
```

Sintassi di espressioni numeriche

Definizione ricorsiva. Un'espressione numerica può essere:

- 1.un'espressione numerica elementare,
- 2.della forma $\langle \text{Expr}_1 \rangle \langle \text{OpBinario} \rangle \langle \text{Expr}_2 \rangle$,
- 3.della forma $(\langle \text{Expr}_1 \rangle) \langle \text{OpBinario} \rangle (\langle \text{Expr}_2 \rangle)$,
- 4.della forma $\langle \text{OpUnarioPrefisso} \rangle \langle \text{Expr}_1 \rangle$,
- 5.della forma $\langle \text{OpUnarioPrefisso} \rangle (\langle \text{Expr}_1 \rangle)$,
- 6.della forma $\langle \text{Expr}_1 \rangle \langle \text{OpUnarioPostfisso} \rangle$,
- 7.della forma $(\langle \text{Expr}_1 \rangle) \langle \text{OpUnarioPostfisso} \rangle$,
- 8.una chiamata ad una funzione che restituisce valori numerici,
- 9.della forma $\langle \text{Expr}_1 \rangle ? \langle \text{Expr}_2 \rangle : \langle \text{Expr}_3 \rangle$,
- 10.della forma $(\langle \text{Expr}_1 \rangle) ? (\langle \text{Expr}_2 \rangle) : (\langle \text{Expr}_3 \rangle)$,

dove $\langle \text{Expr}_1 \rangle$, $\langle \text{Expr}_2 \rangle$ e $\langle \text{Expr}_3 \rangle$ sono espressioni numeriche,
 $\langle \text{OpBinario} \rangle$ è un operatore numerico binario predefinito,
 $\langle \text{OpUnarioPrefisso} \rangle$ è un operatore numerico unario-prefisso predefinito,
 $\langle \text{OpUnarioPostfisso} \rangle$ è un operatore numerico unario-postfisso predefinito.

Operatore condizionale ?= (1/2)

Unico operatore ternario del linguaggio C

Sintassi

`<Expr1> ? <Expr2> : <Expr3>;`

oppure

`(<Expr1>) ? (<Expr2>) : (<Expr3>);`

dove `<Expr1>`, `<Expr2>`, e `<Expr3>` sono espressioni numeriche.

Semantica

- Viene dapprima valutata l'espressione `<Expr1>`.
- Se essa ha un valore non nullo (e, cioè, risulta vera), viene valutata l'espressione `<Expr2>`, ed il risultato ottenuto costituisce il valore dell'intera espressione condizionale.
- Se invece il valore di `<Expr1>` è nullo (e, cioè, `<Expr1>` risulta falsa), viene valutata invece l'espressione `<Expr3>`, ed il risultato ottenuto costituisce il valore dell'intera espressione condizionale.

Operatore condizionale ?= (2/2)

Espressione condizionale:

`<Expr1> ? <Expr2> : <Expr3>;`

Esempio: `(a>b) ? a : b`

Restituisce il massimo fra i valori delle variabili **a** e **b**.

- Se `<Expr2>` e `<Expr3>` sono di tipo diverso, il tipo del risultato è determinato dalle regole di conversione discusse per gli operatori binari distinti da quelli di assegnazione (vedi lezione 4). Ad esempio, se **a** è un **float** e **b** è un **int** il risultato di `(a>b) ? a : b` è di tipo **float**.
- La precedenza dell'operatore ternario `?=` è molto bassa. È superiore solo a quella degli operatori di assegnazione.

Precedenza e associatività

Precedenza e associatività degli operatori discussi finora.

Gli operatori sono mostrati dall'alto in basso in ordine decrescente di precedenza.

Operatori	Associatività	Tipo
<code>++ (postfisso) -- (postfisso)</code>	da destra a sinistra	postfisso
<code>+ - ! ++ (prefisso) -- (prefisso) (tipo)</code>	da destra a sinistra	unario
<code>* / %</code>	da sinistra a destra	moltiplicativo
<code>+ -</code>	da sinistra a destra	additivo
<code>< <= > >=</code>	da sinistra a destra	relazionale
<code>== !=</code>	da sinistra a destra	di uguaglianza
<code>&&</code>	da sinistra a destra	AND logico
<code> </code>	da sinistra a destra	OR logico
<code>? :</code>	da destra a sinistra	condizionale
<code>= += -= *= /= %=</code>	da destra a sinistra	di assegnazione

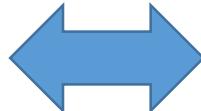
Curiosità (1/2)

Le istruzioni di selezione possono essere espresse in termini dell'istruzione while.

Prima mostriamo che l'istruzione **if-else** può essere espressa in termini dell'istruzione di selezione singola **if**.

Per ogni occorrenza dell'istruzione **if-else** nel programma introduciamo una nuova variabile intera, chiamiamola **z**, e allora operiamo la seguente sostituzione:

```
if(<condizione>
{
    <corpo 1>
}
else
{
    <corpo 2>
}
```



```
int z:= <condizione>;
if(z)
{
    <corpo 1>
}
if(!z)
{
    <corpo 2>
}
```

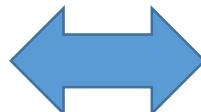
Curiosità (2/2)

Le istruzioni di selezione possono essere espresse in termini dell'istruzione while.

Ora mostriamo che l'istruzione di selezione singola **if** può essere espressa in termini dell'istruzione **while**.

Per ogni occorrenza dell'istruzione **if** nel programma introduciamo una nuova variabile intera, chiamiamola **z**, e allora operiamo la seguente sostituzione:

```
if(<condizione>
{
    <corpo>
}
```



```
int z:= <condizione>;
while(z)
{
    z = !z;
    <corpo>
}
```

Esercizi

1. Scrivere un programma che, dati un intero (inseriti da tastiera), stampi a video una per una le sue cifre decimali (a partire dall'ultima) e, infine, il numero complessivo di cifre.
2. Si scriva un programma che ad ogni passo chiede all'utente di inserire un intero positivo e stampa il prodotto degli interi ricevuti in input fino a quando o l'utente inserisce un valore non valido (sentinella) o si raggiunge un overflow. Distinguere le due situazioni di terminazione (utilizzare le costanti del file limits.h) stampando un opportuno messaggio.
3. Scrivere un programma che, dato un intero positivo N, stampi a video il prodotto degli interi positivi dispari minori o uguali a N, se è possibile rappresentare tale prodotto in termini di un valore di tipo int. E altrimenti segnala il conseguente overflow.

Esercizi

4. Scrivere un programma che, dato in input un intero (inserito da tastiera), calcoli e stampi tutti i suoi divisori. Si ricorda che un intero d è un divisore di un intero n se esiste un intero k tale quale $n = d \times k$.
5. Si scriva un programma che, dato in input un intero positivo (inserito da tastiera), calcoli e stampi a video il suo fattoriale $n!$. Si ricorda che $n!$ è definito per induzione su n come segue: se $n=1$, allora $n!=1$; altrimenti, $n!=n \times (n-1)!$.
6. Scrivere un programma che, dati due interi a e b in input (inseriti da tastiera), calcoli e stampi a^b .

Lezione 7

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 7: Strutture di controllo (seconda parte)

- Istruzione di selezione switch.
- Istruzione di iterazione for.
- Istruzione di iterazione do...while.
- Istruzioni break e continue.
- Esercizi.

Flusso di esecuzione nel linguaggio C

Nel linguaggio C, le **istruzioni atomiche**, cosiddette perchè non sono decomponibili (corrispondono ad azioni elementari), che compaiono nel corpo di una funzione possono essere classificate in:

- Istruzioni di dichiarazione (ed inizializzazione) di variabili.
- Istruzioni di assegnazione.
- Chiamate di funzioni.

Normalmente, le istruzioni sono eseguite una dopo l'altra nell'ordine in cui sono scritte (**modalità di esecuzione sequenziale**). Il linguaggio C fornisce varie istruzioni non-atomiche, chiamate in generale **istruzioni di controllo**, per specificare un'ordine di esecuzione non sequenziale (**modalità di traferimento del controllo**) :

- **Istruzioni di selezione o condizionali:** per specificare una ramificazione condizionale nel flusso di controllo.
- **Istruzioni di iterazione:** per consentire la ripetizione condizionale di una sequenza di azioni.

Istruzioni di selezione

Consentono di specificare che sequenze di istruzioni debbano essere eseguite o meno in dipendenza del soddisfacimento o meno di una certa condizione (**ramificazione condizionale nel flusso di controllo**).

- Istruzione di selezione semplice **if**: un'alternativa.
- Istruzione di selezione doppia **if-else**: due alternative.
- Istruzione di selezione plurima **switch**: multiple alternative.

NOTA: sono tutte istruzioni ad un solo ingresso e a una sola uscita.

Espressioni numeriche costanti

Espressioni numeriche dove ogni sottoespressione elementare è una costante numerica (costante intera, costante reale, o costante carattere).

Il valore di un' espressione numerica costante è sempre lo stesso (indipendente dallo stato di elaborazione). Esso può essere determinato a tempo di compilazione.

Esempi:

$4 * 3.4 / 2$

$3.5 + 'a' / 3 == ! 89$

Istruzione switch (1/5)

- Istruzione di controllo basata su scelte multiple.
- Controlla se un'espressione numerica (**espressione di controllo**) assume un valore all'interno di un certo insieme finito di costanti numeriche **distinte** (valori di espressioni numeriche costanti), dove ad ogni costante è associata una sequenza di istruzioni.
- Se il valore dell'espressione di controllo coincide con una delle costanti numeriche, vengono eseguite le istruzioni corrispondenti. In caso contrario, vengono eseguite le istruzioni associate alla **scelta di default** (se presente), e non viene eseguita nessuna istruzione altrimenti.
- Al termine dell'esecuzione dell'istruzione, l'esecuzione procede con l'istruzione successiva all'istruzione **switch**.

Istruzione switch (2/5)

Sintassi

```
switch(<esp>)
{
    case <esp_const1>:
        <corpo1>
        .
        .
        .
    case <esp_constN>:
        <corpoN>
    default:
        <corpoN+1>
}
```

<esp> è l'**espressione di controllo**: generica espressione numerica.

Caso etichettato dall'espressione numerica costante **<esp_const₁>** dove **<corpo₁>** è la sequenza associata di istruzioni (eventualmente vuota).

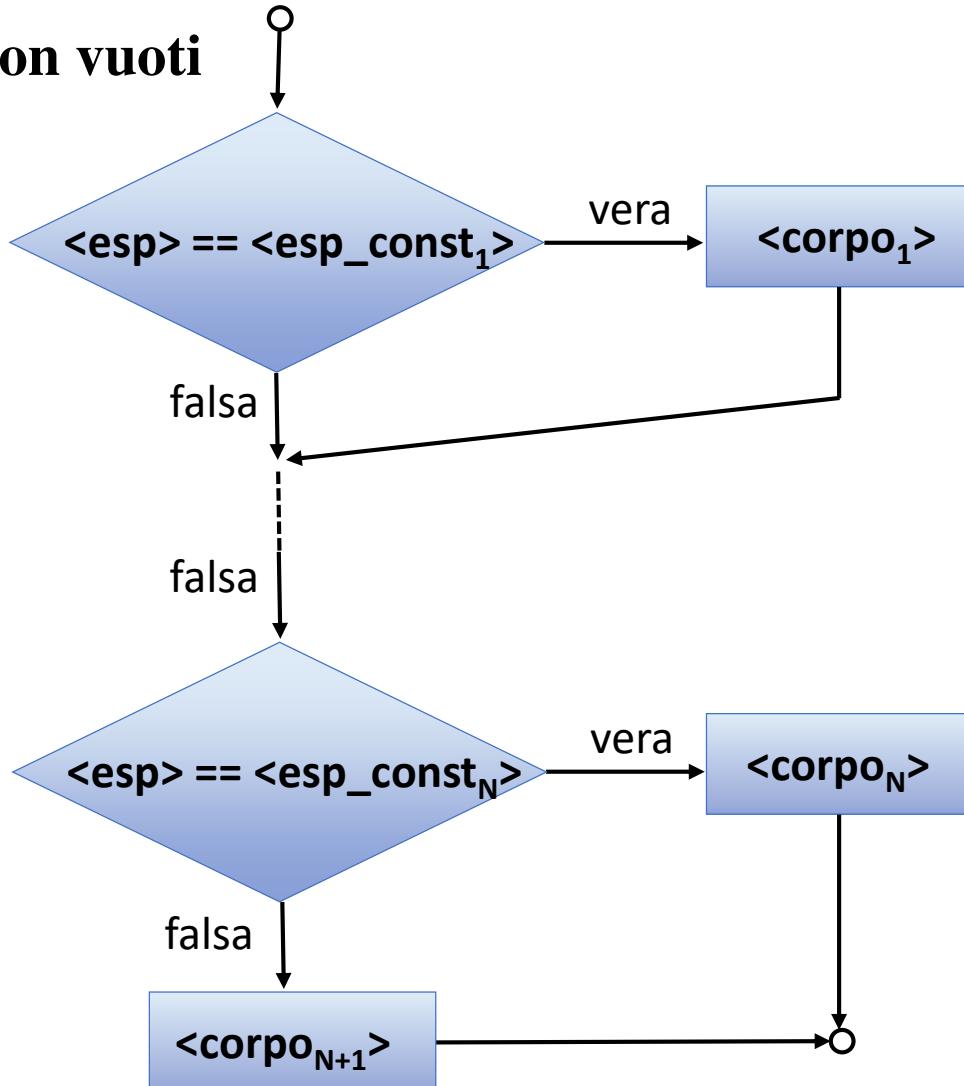
Caso di default, **opzionale**, dove **<corpo_{N+1}>** è la sequenza associata di istruzioni (eventualmente vuota).

Espressioni numeriche costanti associate a distinti casi devono assumere valori diversi. Altrimenti, il compilatore genera un errore. Casi con corpi vuoti vengono utilizzati per associare a più casi le stesse istruzioni.

Istruzioni switch(3/5)

Semantica con corpi non vuoti

```
switch(<esp>)
{
    case <esp_const1>:
        <corpo1>
        .
        .
        .
    case <esp_constN>:
        <corpoN>
    default:
        <corpoN+1>
}
```



Istruzione switch (4/5)

- L'espressione di controllo viene confrontata con **ciascuna** delle espressioni costanti associate alle varie **etichette case**.
- Se il valore dell'espressione di controllo coincide con il valore dell'espressione numerica costante associata ad un'etichetta case viene eseguita l'associata sequenza di istruzioni. Altrimenti, viene eseguita la sequenza di istruzioni associata all'**etichetta default**.
- Dal momento che le espressioni costanti associate alle diverse **etichette case** hanno valori diversi, vi sarà al più un'etichetta case la cui sequenza di istruzioni verrà eseguita.
- **Cascata di etichette case con corpo vuoto:** ad un'etichetta case con corpo vuoto viene associata la sequenza di istruzioni della prima etichetta che la segue (se esiste) avente corpo non vuoto.
- Per forzare l'uscita dall'istruzione switch una volta che la sequenza di azioni associata ad un'etichetta case è stata eseguita, è possibile utilizzare l'**istruzione break**.

Istruzione switch (5/5)

Sintassi con break

```
switch(<esp>)
{
    case <esp_const1>:
        <corpo1>
        break;
    .
    .
    .
    case <esp_constN>:
        <corpoN>
        break;
    default:
        <corpoN+1>
}
```

<esp> è l'**espressione di controllo**: generica espressione numerica.

Caso etichettato dall'espressione numerica costante **<esp_const₁>** dove **<corpo₁>** è la sequenza associata di istruzioni (eventualmente vuota).

Caso di default, **opzionale**, dove **<corpo_{N+1}>** è la sequenza associata di istruzioni (eventualmente vuota).

Esempio istruzione switch

Dato un carattere (inserito da tastiera), distinguere i casi in cui il carattere rappresenta una cifra decimale pari, una cifra decimale dispari, o un qualsiasi altro carattere.
Stampare un messaggio opportuno.

```
char carattere = '0';
printf("Inserisci un carattere!\n");
scanf("%c",&carattere);

switch(carattere)
{
    //il carattere è una cifra decimale pari
    //(cascata di etichette case)
    case '0': case '2': case '4': case '6': case '8':
        printf("Il carattere e' una cifra decimale pari!\n");
        break;

    //il carattere è una cifra decimale dispari
    //(cascata di etichette case)
    case '1': case '3': case '5': case '7': case '9':
        printf("Il carattere e' una cifra decimale dispari!\n");
        break;

    default:
        printf("Il carattere non e' una cifra decimale!\n");
}
```

Istruzioni di iterazione

Consentono di ripetere l'esecuzione di una sequenza di istruzioni finchè una certa condizione rimane soddisfatta.

- Istruzione di iterazione **while** (detta anche **ciclo while**).
- Istruzione di iterazione **for**.
- Istruzione di iterazione **do...while**.

NOTA: sono tutte istruzioni ad un solo ingresso e a una sola uscita.

Modalità d'iterazione (1/2)

- **Iterazione controllata da contatore:** chiamata anche *iterazione definita*, perchè sappiamo *in anticipo* quante volte sarà eseguito il ciclo:
 - Viene utilizzata una variabile di controllo per contare il numero di iterazioni.
 - La variabile di controllo è incrementata (di solito di 1) ogni volta che viene eseguita un'iterazione del ciclo.
 - Quando il valore della variabile di controllo indica che è stato eseguito il numero corretto di iterazioni, il ciclo termina e l'esecuzione continua con l'istruzione dopo l'istruzione d'iterazione.

Modalità d'iterazione (2/2)

- **Iterazione controllata da sentinella:** chiamata anche *iterazione indefinita*, perchè non si sa in anticipo quante volte sarà eseguito il ciclo:
 - Viene utilizzata quando il corpo del ciclo comprende istruzioni che leggono dati ad ogni iterazione.
 - Il valore sentinella corrisponde alla condizione di terminazione del ciclo ed indica la ‘fine dei dati in ingresso’.
 - La sentinella è inserita dopo che tutti i dati regolari sono stati forniti al programma e deve essere distinta dai dati regolari.

Iterazione controllata da contatore

L'iterazione controllata da contatore richiede:

- dichiarazione di una variabile di controllo e sua inizializzazione,
- incremento (o decremento) della variabile di controllo ad ogni iterazione,
- condizione che verifica il valore finale della variabile di controllo (cioè se il ciclo deve continuare).

Esempio: stampiamo gli interi da 1 a 10 tramite iterazione **while** controllata da contatore.

```
int main(void)
{
    unsigned int counter = 1; // inizializzazione

    while ( counter <= 10 ) { // condizione di iterazione
        printf ( "%u\n", counter );
        ++counter; // incremento
    }
}
```

Istruzione d'iterazione **for**

L'istruzione di iterazione **for** comprende tutti i dettagli dell'iterazione controllata da contatore. Come tutte le istruzioni di controllo consiste di un **corpo di istruzioni** eseguito ad ogni iterazione del ciclo **for** e, addizionalmente:

- **Parte di inizializzazione:** viene eseguita solo all'inizio dell'esecuzione dell'istruzione **for**. Inizializza e, possibilmente, dichiara le variabili di controllo.
- **Condizione di continuazione del ciclo:** se la condizione è vera, viene eseguito il corpo del ciclo. Altrimenti, l'esecuzione dell'istruzione **for** termina ed il controllo è trasferito all'istruzione che segue quella **for** nel programma.
- **Parte di incremento:** al termine di ogni iterazione viene eseguita la parte di incremento che incrementa (o decrementa) le variabili di controllo, dopodichè il ciclo continua a partire dalla valutazione della condizione di continuazione.

Sintassi istruzione for (1/6)

Sintassi

for(<inizializzazione>; <condizione>;<incremento>)

{

<corpo>

}

- **<inizializzazione>**: rappresenta la fase di inizializzazione del ciclo.
- **<condizione>**: espressione numerica generica che rappresenta la condizione di continuazione del ciclo.
- **<incremento>**: rappresenta la parte di incremento del ciclo.
- **<corpo>**: rappresenta una sequenza di istruzioni (istruzioni atomiche o di controllo), eseguite ad ogni iterazione.

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione di iterazione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Sintassi istruzione for (2/6)

Sintassi

```
for(<inizializzazione>; <condizione>;<incremento>)
{
    <corpo>
}
```

Fase di inizializzazione: **<inizializzazione>** è una sequenza arbitraria di istruzioni atomiche dove

- il terminatore ‘;’ di ogni non ultima istruzione nella sequenza è sostituito dall’operatore virgola ‘,’. Per l’ultima istruzione nella sequenza il terminatore ‘;’ è omesso.
- Ogni dichiarazione di variabile nell’inizializzazione deve essere inizializzata.

Esempio di inizializzazione: int x=0, int y=5

NOTA: usualmente la parte di inizializzazione contiene dichiarazioni inizializzate di variabili semplici (evitare chiamate di funzioni). La parte di inizializzazione può essere omessa!

Sintassi istruzione for (3/6)

Sintassi

```
for(<inizializzazione>; <condizione>;<incremento>)
{
    <corpo>
}
```

Condizione di continuazione del ciclo: **<condizione>** è un'espressione numerica generica. Tale condizione può essere omessa.

Se viene omessa l'espressione **<condizione>**, il C considera che la condizione di continuazione del ciclo sia *vera*, generando così un *ciclo infinito*.

Esempio di condizione: `x<= 10 && y>= 20`

Sintassi istruzione for (4/6)

Sintassi

```
for(<inizializzazione>; <condizione>;<incremento>)
{
    <corpo>
}
```

Parte di incremento: **<incremento>** è una sequenza arbitraria di istruzioni atomiche dove

- il terminatore ‘;’ di ogni non ultima istruzione nella sequenza è sostituito dall’operatore virgola ‘,’. Per l’ultima istruzione nella sequenza il terminatore ‘;’ è omesso.
- La sequenza non deve contenere dichiarazioni di variabili.

Esempio di incremento: x++, -- y

NOTA: usualmente la parte di incremento contiene solo istruzioni di aggiornamento delle variabili inizializzate (evitare chiamate di funzioni). La parte di incremento può essere omessa!

Sintassi istruzione for (5/6)

Sintassi

```
for(<inizializzazione>; <condizione>;<incremento>)
{
    <corpo>
}
```

Come abbiamo visto precedentemente, tutte le tre parti (inizializzazione, condizione, ed incremento) che compongono l'intestazione del ciclo **for** sono opzionali.

In ogni caso, i due punti e virgola ‘;’ alla sinistra e alla destra di **<condizione>** devono essere sempre presenti.

Istruzione di iterazione for (6/6)

Sintassi semplificata quando il corpo consiste di una sola istruzione

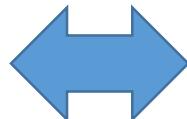
for(<inizializzazione>; <condizione>;<incremento>)

<corpo>

<corpo> deve consistere di un'unica istruzione (atomica o di controllo).

Esempio:

```
for(int x=0; x<= 10; x++)  
{  
    z += x;  
}
```

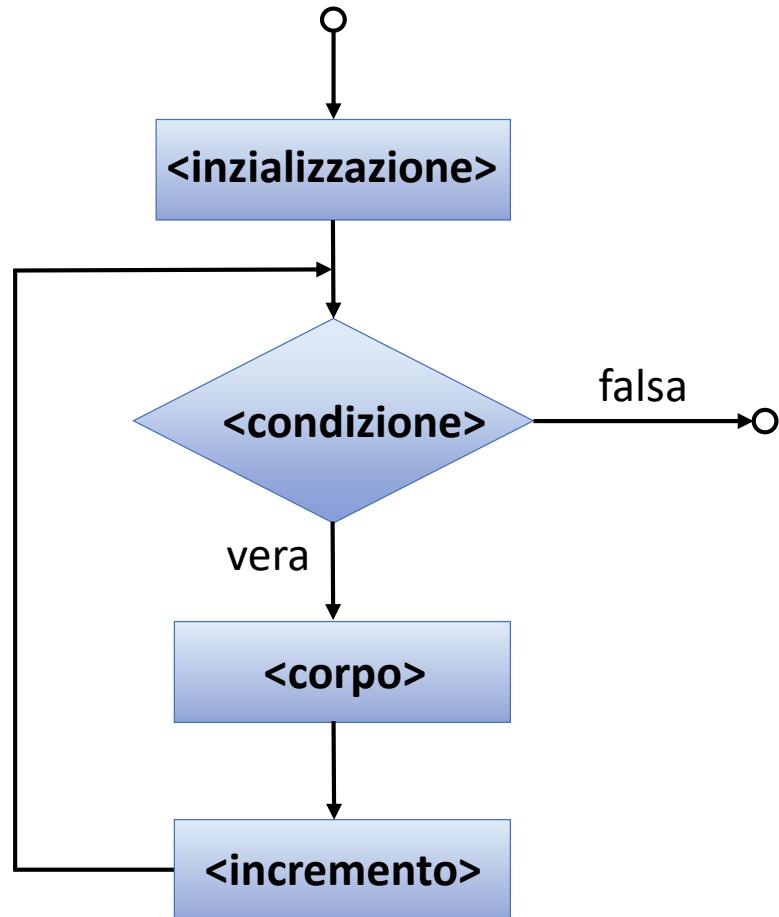


```
for(int x=0; x<= 10; x++)  
    z += x;
```

Semantica istruzione di iterazione for

```
for(<inizializzazione>;  
    <condizione>;<incremento>)  
{  
    <corpo>  
}
```

1. Viene eseguita (**<inizializzazione>**)
2. Se **<condizione>** è vera (e, cioè, il valore è diverso da zero), viene eseguito prima il corpo di istruzioni **<corpo>** e poi la parte **<incremento>** e si ripete il punto 2.
3. Se **<condizione>** è falsa (e, cioè, il valore è uguale a zero), il ciclo termina e viene eseguita l'istruzione successiva all'istruzione **for**.



Esempio istruzione d'iterazione for (1/2)

Esempio: stampiamo gli interi da 1 a 10.

Tramite iterazione while:

```
unsigned int counter = 1; // inizializzazione

while ( counter <= 10 ) { // condizione di iterazione
    printf ( "%u\n", counter );
    ++counter; // incremento
}
```

Tramite iterazione for:

```
// inizializzazione, condizione dell'iterazione e incremento
// sono tutti inclusi nell'intestazione dell'istruzione for.
for (unsigned int counter = 1; counter <= 10; ++counter) {
    printf("%u\n", counter);
}
```

Esempio istruzione d'iterazione for (2/2)

Esempio: stampiamo gli interi pari da 0 a 10.

```
| for(unsigned int i=0; i<=10; i +=2)  
|     printf("%d\n",i);
```

Esempio: stampiamo gli interi pari da 0 a 10 in ordine inverso.

```
| for(unsigned int i=10; i>=0; i -=2)  
|     printf("%d\n",i);
```

Istruzione d'iterazione do...while

L'istruzione di iterazione **do..while** è simile all'istruzione di iterazione while:

- Nell'istruzione **while** la condizione di continuazione del ciclo è verificata all'*inizio* del ciclo *prima* dell'esecuzione del corpo del ciclo.
- Nell'istruzione **do..while** la condizione di continuazione del ciclo è verificata *dopo* l'esecuzione del corpo del ciclo. Pertanto, il corpo del ciclo sarà sempre eseguito *almeno una volta*.
- Quando un'istruzione **do..while** termina, l'esecuzione prosegue con l'istruzione dopo l'istruzione **do..while**.

Istruzione di iterazione do..while (1/3)

Sintassi

```
do  
{  
    <corpo>  
}  
while (<condizione>);
```

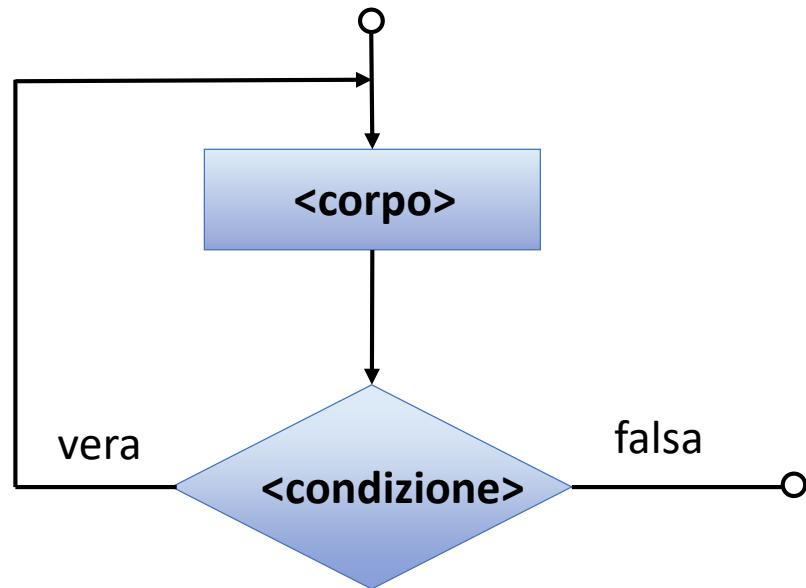
dove **<condizione>** è un'espressione numerica generica, e **<corpo>** è una sequenza arbitraria di istruzioni (istruzioni atomiche o istruzioni di controllo).

Buona norma: far rientrare a destra il testo dell'intero corpo dell'istruzione di iterazione con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Nota: usare sempre le parentesi graffe che racchiudono il corpo delle istruzioni anche se **<corpo>** consiste di un'unica istruzione. Questo per evitare confusione nell'individuazione della condizione di continuazione del ciclo.

Istruzione di iterazione do..while (2/3)

```
do  
{  
    <corpo>  
}  
while(<condizione>);
```



Semantica

- Viene eseguito il corpo `<corpo>` dell'istruzione e, successivamente, viene valutata l'espressione `<condizione>`.
- Se l'espressione `<condizione>` è vera (e, cioè, il valore restituito è diverso da 0), l'istruzione `do..while` viene eseguita nuovamente (nuova iterazione del ciclo). Altrimenti, l'istruzione `do..while` termina.
- Al termine dell'esecuzione dell'istruzione `do..while`, l'esecuzione procede con l'istruzione successiva all'istruzione `do..while`.

Istruzione di iterazione do..while (3/3)

Esempio: Calcolare la media dei voti di un gruppo di studenti.

```
int voto =-1;
int contatore =0;
int totale =0;
int cond =0;

do
{
    printf("Inserisci un voto.\n");
    cond = scanf("%d",&voto) ==1 && voto >=0;
    if(cond !=0)
    {
        contatore++;
        totale += voto;
    }

}while(cond !=0);

if(contatore >0)
    printf("La media dei voti e': %.1f", (float)totale/contatore);
else
    printf("Non e' stato inserito alcun voto!\n");
```

Istruzioni per alterare il flusso di controllo (1/3)

- Istruzione **break**: l'istruzione **break**; viene utilizzata all'interno del corpo di un'istruzione d'iterazione (ciclo *while*, ciclo *for*, o ciclo *do..while*) e la sua esecuzione causa l'immediata uscita dal ciclo. Come già abbiamo visto, può essere anche utilizzata all'interno di un'istruzione *switch* per terminare l'esecuzione dell'istruzione *switch*.
 - L'istruzione **break**; provoca l'uscita dall'istruzione *switch* oppure dal ciclo più interno che la contiene.

Istruzioni per alterare il flusso di controllo (2/3)

Esempio utilizzo istruzione break: verificare che un intero positivo è un numero primo.

```
unsigned int N = 0;
int cond =1;

while(cond)
{
    printf("Inserisci un numero intero non negativo.\n");
    cond = scanf("%u",&N);

    int isPrimo = 1;
    unsigned int i;
    for(i=2;i<N;i++)
    {
        if(N % i == 0)
        {
            isPrimo = 0;
            printf("Il numero inserito non e' primo ed il suo piu' piccolo divisore e': %d\n",i);
            break;
        }
        else if(i*i>= N)
            break;
    }

    if(N==0 || N==1)
        printf("Il numero inserito non e' primo \n");
    else if(isPrimo)
        printf("Il numero inserito e' un numero primo \n");
}
```

Istruzioni per alterare il flusso di controllo (3/3)

- Istruzione **continue**: l'istruzione **continue**; viene utilizzata all'interno del corpo di un'istruzione d'iterazione (ciclo *while*, ciclo *for*, o ciclo *do..while*) per terminare immediatamente l'iterazione corrente e passare all'iterazione successiva.
 - Nel caso del ciclo *for*, il controllo passa alla parte di incremento prima di iniziare una nuova iterazione.
 - L'istruzione **continue**; provoca la terminazione dell'iterazione corrente del ciclo più interno che la contiene.
 - L'istruzione **continue**; viene spesso utilizzata quando la parte successiva del corpo di un ciclo è particolarmente complicata; in questo caso, infatti, invertire un test e creare un altro livello di indentazione potrebbe condurre alla stesura di codice troppo nidificato.

Esercizi

1. Scrivere un programma che, dati un intero in input (inserito da tastiera), testi se questo è palindromo o meno e stampi conseguentemente un opportuno messaggio.
2. Scrivere un programma che, dato un intero N non negativo in input (inserito da tastiera), stampi gli N primi termini della serie di Fibonacci. Ricordiamo che i due primi termini della serie di Fibonacci sono $f_0 = f_1 = 1$, e poi per ogni $N \geq 2$, $f_N = f_{N-1} + f_{N-2}$.
3. Scrivere un programma che dato un intero positivo in input, lo stampi cifra per cifra in binario, a partire dalla cifra meno significativa, ed, infine stampi il numero delle cifre binarie.
4. Si considerino due numeri rappresentati in binario su N bit, dove N è inserito da tastiera. I due numeri sono inseriti da tastiera un bit alla volta a partire dal bit meno significativo, e, cioè ad ogni passo vengono inseriti i due bit alla posizione corrente. Si scriva un programma per eseguire la somma dei due numeri e che stampi tale somma in decimale.

Esercizi

4. Scrivere un programma per la rappresentazione del triangolo di Floyd. Il triangolo di Floyd è un triangolo rettangolo che contiene numeri naturali, definito riempiendo le righe del triangolo con numeri consecutivi e partendo da 1 nell'angolo in alto a sinistra. Si consideri ad esempio il caso N=5. Il triangolo di Floyd è il seguente:

```
1  
2 3  
4 5 6  
7 8 9 10  
11 12 13 14 15
```

Il programma riceve da tastiera un numero intero N. Il programma visualizza le prime N righe del triangolo di Floyd.

Suggerimento. Si osserva che il numero di valori in ogni riga corrisponde all'indice della riga: 1 valore sulla prima riga, 2 sulla seconda, 3 sulla terza.

Lezione 11

Laura Bozzelli
a.a. 2020/2021

Sommario - Lezione 11: Tipi di dati definiti da utente

- Strutture.
- Unioni.
- Tipi enumerativi.
- Introduzione agli array.
- Array monodimensionali.
- Passaggio per riferimento degli array.

Tipi definiti da utente: strutture

- Nel linguaggio C, una **struttura**, detta anche aggregato, è una collezione di dati (variabili), di uno o più tipi, raggruppate sotto un unico nome.
- Le strutture possono contenere variabili di tipi (predefiniti o definiti dall'utente) differenti.
- Aiutano ad organizzare dati complessi in quanto consentono di trattare come un unico oggetto (dato aggregato) un insieme di variabili correlate.
- Esempio: i dati relativi ad uno studente (nome, cognome, numero di immatricolazione, media esami, ecc..) possono essere raggruppati in un'unica struttura.

Definizione di un tipo struttura (1/5)

Sintassi

```
struct <nome tipo>
{
    <dichiarazione variabile 1>
    ....
    <dichiarazione variabile N>
};
```

Dichiarazioni di variabili racchiuse tra parentesi graffe, terminante con un punto e virgola, e avente come intestazione la parola chiave **struct** seguita da un identificatore di tipo.

- **<nome tipo>**: è qualsiasi identificatore valido (come visto per gli identificatori delle variabili semplici o dei nomi di funzione). Chiamato anche **tag**, rappresenta l'**identificatore del tipo di struttura**. Similmente agli identificatori (parole chiavi) dei tipi predefiniti (int, char, float,ecc..), esso viene utilizzato per dichiarare variabili del dato tipo.

Definizione di un tipo struttura (2/5)

Sintassi

```
struct <nome tipo>
{
    <dichiarazione variabile 1>
    ....
    <dichiarazione variabile N>
};
```

- **<dichiarazione variabile 1> <dichiarazione variabile N>** : dichiarazioni di variabili (di tipo predefinito o definito dall'utente) senza inizializzazione e senza specificatori di classi di memoria.
- Le variabili dichiarate entro le parentesi nella definizione di un tipo struttura sono dette **membri** o **campi** della struttura.

Buona norma: far rientrare a destra il testo delle dichiarazioni di variabili con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Definizione di un tipo struttura (3/5)

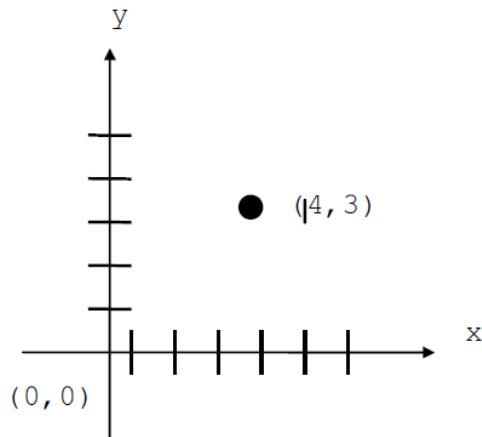
Sintassi

```
struct <nome tipo>
{
    <dichiarazione variabile 1>
    ....
    <dichiarazione variabile N>
};
```

- I **membri** di un tipo di struttura devono avere nomi differenti, ma due tipi di strutture differenti possono contenere membri dello stesso nome senza che ciò crei un conflitto.
- Ogni definizione di un tipo struttura deve terminare con un punto e virgola.
- Una definizione di tipo struttura definisce **un tipo**. Non viene allocata alcuna memoria in corrispondenza di una definizione di tipo. La memoria viene allocata solo tramite dichiarazioni (alias definizioni) di variabili di quel tipo.

Definizione di un tipo struttura (4/5)

Esempio: un qualsiasi punto del piano cartesiano è una coppia ordinata di numeri reali.



Possiamo utilizzare il seguente tipo di struttura per rappresentare tali punti.

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};
```

Definizione di un tipo struttura (5/5)

- Una definizione di tipo struttura può comparire nel corpo di una funzione. In questo caso, il suo **campo di azione o scope** va dal punto in cui tale definizione è scritta al termine del corpo della funzione.
- Quando una definizione di tipo struttura compare all'esterno di una definizione di funzione, il suo **campo di azione o scope** va dal punto della definizione fino alla fine del file in cui la definizione si trova. Per consentire l'accesso in più file sorgenti, conviene porre la definizione di un tipo struttura in un file header ed includere il file tramite la direttiva **#include** del preprocessore.

Dichiarazione variabili di tipo struttura (1/3)

Sintassi per tipi di struttura con identificatore di tipo:

struct <nome tipo struttura> <nome var 1>, ..., <nome var N>;

- **<nome tipo struttura>**: identificatore della definizione di tipo struttura.
- **<nome var 1>, ..., <nome var N>**: nomi di variabili (qualsiasi identificatore valido).

La dichiarazione di variabile alloca una area contigua di memoria per contenere tutti i **membri** di quella variabile.

Esempio:

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};
```

```
struct Punto punto1, punto2;
struct Punto punto3;
```

Dichiarazione variabili di tipo struttura (2/3)

Variabili di tipo struttura possono essere anche dichiarate immediatamente dopo la definizione dell'associato tipo struttura e prima del terminatore punto e virgola.

```
struct Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
} punto1, punto2;
```

Dichiarazione variabili di tipo struttura (3/3)

Una definizione di tipo struttura può essere anche **anonima** e, cioè, non contenere alcun identificatore di tipo. In questo caso, le variabili di quel tipo possono essere dichiarate solo immediatamente dopo la definizione di tipo struttura.

Esempio:

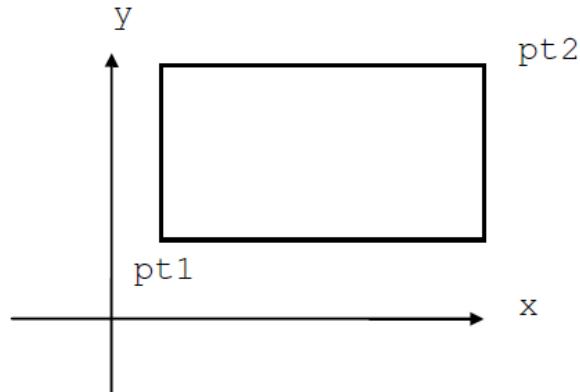
```
struct
{
    float x; //coordinata Lungo l'asse X
    float y; //coordinata Lungo l'asse Y
} punto1, punto2;
```

NOTA: una definizione di tipo struttura **anonima** può essere parte di una definizione di struttura con identificativo per dichiarare campi di tipo struttura con tipo anonimo.

Strutture annidate

Membri di una struttura possono essere a loro volta variabili di tipo struttura (e, cioè, le strutture possono essere arbitrariamente annidate).

Esempio 2: un rettangolo nel piano cartesiano può essere identificato da due punti, il punto corrispondente all'angolo superiore-destro ed il punto corrispondente all'angolo inferiore-sinistro.



Possiamo utilizzare il seguente tipo di struttura per rappresentare rettangoli nel piano.

```
struct Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};
```

Inizializzazione variabili di tipo struttura (1/2)

Dichiarazione variabile di tipo struttura con inizializzazione:

struct <nome tipo struttura> <nome var> = <inizializzazione>;

Sintassi <inizializzazione>: rappresenta la parte di inizializzazione.

{ <Init 1>,,<Init N>}

- N deve essere non superiore al numero dei membri della struttura.
- <Init i> inizializza l'i-esimo membro ($1 \leq i \leq N$): per un membro di tipo numerico (semplice), è un'espressione numerica generica. Per un membro di tipo struttura è a sua volta un'espressione di inizializzazione oppure il nome di una variabile dello stesso tipo, oppure una chiamata a funzione restituente un dato di quel tipo.
- Se N è inferiore al numero di membri della struttura, i restanti membri sono inizializzati a zero (per membri di tipo numerico).
- Per **dichiarazioni di variabili globali o statiche**, le espressioni numeriche utilizzate devono essere costanti (i valori di inizializzazione devono essere noti a tempo di compilazione).

Inizializzazione variabili di tipo struttura (2/2)

```
struct Punto
{
    float x; //coordinata Lungo l'asse X
    float y; //coordinata Lungo l'asse Y
};

struct Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

struct Rettangolo rett={{1.34,2},{2.34,5.1}};

struct Punto RitornaPunto(); //Prototipo di funzione c
                            //che restituisce una struttura Punto

void Prova()
{
    float x = 3.5, y = 2.7;
    struct Punto p1 = {x,y+1};
    struct Rettangolo ret = {p1, RitornaPunto()};
    .....
}
```

Utilizzo dati di tipo struttura (1/3)

Le sole operazioni valide eseguibili su strutture sono le seguenti:

- Assegnazione a variabili di tipo struttura.

<nome variabile> = <espressione>;

- dove **<espressione>** può essere o un nome di variabile struttura dello stesso di tipo di **<nome variabile>** oppure una chiamata di funzione che restituisce come risultato una struttura dello stesso tipo di **<nome variabile>** .
- Il ‘valore aggregato’ dell’espressione viene assegnato alla variabile struttura a sinistra dell’istruzione di assegnazione.

```
struct Punto RitornaPunto(); //Prototipo di funzione c  
//che restituisce una struttura Punto  
  
void Prova()  
{  
    struct Punto p1,p2;  
    p1 = RitornaPunto();  
    p2 = p1;  
}
```

Utilizzo dati di tipo struttura (2/3)

- Passaggio di argomenti di tipo struttura alle funzioni e la restituzione di valori struttura dalle funzioni. Le funzioni possono avere come parametri formali variabili sia di tipo predefinito che di tipo definito dall'utente e restituire valori di tipo arbitrario. Il passaggio degli argomenti è sempre per valore: il valore dell'argomento di tipo struttura viene copiato nel parametro formale di tipo struttura.

Esempio:

```
//Prototipo di funzione avente strutture come parametri formali
//che restituisce una struttura
struct Punto SommaPunti(struct Punto p1,struct Punto p2);
```

- Puntatori a strutture (lo vedremo nel seguito).

Utilizzo dati di tipo struttura (3/3)

- Accesso ai campi (membri) di una variabile di tipo struttura tramite l'**operatore di campo di struttura** ‘.’.

<nome variabile>.<nome campo>

- identifica la locazione di memoria dell’associato campo della variabile struttura. Se tale campo è a sua volta una struttura, si può far seguire all’espressione precedente l’operatore ‘.’ per accedere in modo annidato ai suoi membri.

```
struct Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

float CalcolaArea(struct Rettangolo rect)
{
    return (rect.pt2.y-rect.pt1.y) * (rect.pt2.x-rect.pt1.x);
}
```

Esempi utilizzo dati di tipo struttura

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};

struct Punto CreaPunto(float x,float y)
{
    struct Punto temp;
    temp.x=x;
    temp.y=y; //Non esiste conflitto fra i nomi dei parametri e quelli
              //della struttura; al contrario l'omonimia evidenzia la
              //relazione tra essi.
    return temp;
}

struct Punto AggiungiPunti(struct Punto p1, struct Punto p2)
{
    //Il passaggio di argomenti è sempre per valore.
    //La modifica del parametro formale p1 non si ripercuote
    //sull'argomento strutturato passato in input dal chiamante
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Tipi definiti da utente: unioni

- Nel linguaggio C, un’**unione** è una collezione di dati (**membri** o **campi**) che *condividono lo stesso spazio di memoria*.
- Per diverse situazioni in un programma, alcune variabili possono non essere rilevanti mentre altre possono esserlo, e così un’unione fa condividere spazio di memoria. Come le strutture, i campi di un’unione possono essere di tipo diverso. Il numero di byte necessario per memorizzare un’unione deve essere sufficiente a contenere il campo più grande.
- Si può far riferimento soltanto ad un campo alla volta e di conseguenza ad un solo tipo di dati alla volta. È responsabilità del programmatore tenere traccia del campo attualmente utilizzato.

Utilizzo di tipi unione

- La definizione di un tipo unione è analoga alla definizione di un tipo struttura con l'unica differenza che viene utilizzata la parola chiave **union** invece di **struct**.
- Stesso discorso si applica alle dichiarazioni di variabili. C'è, comunque, un'importante differenza nelle dichiarazioni di variabili con inizializzazione. Per le unioni, è **possibile inizializzare solo il primo campo**.
- Similmente alle strutture, le operazioni che è possibile effettuare sulle unioni sono:
 - Assegnazione a variabili di tipo unione.
 - Passaggio di argomenti di tipo unione alle funzioni e la restituzione di valori unione dalle funzioni.
 - Puntatori a unione (lo vedremo nel seguito).
 - Accesso ai campi (membri) di una variabile di tipo unione tramite l'**operatore di campo ‘.’**.

Esempi utilizzo dati di tipo unione (1/2)

Assumiamo che un oggetto grafico possa essere o un punto o un rettangolo. Utilizziamo una struttura con due campi per rappresentare un oggetto grafico: il primo è un flag indicante se l'oggetto è un punto o un rettangolo. Il secondo campo è un'unione avente due campi: un campo di tipo **Rettangolo** e un campo di tipo **Punto**.

```
struct OggettoGrafico
{
    // è maggiore di zero se
    // l'oggetto grafico è un rettangolo
    int flagRettangolo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
        struct Punto pt;
        struct Rettangolo rect;
    } oggetto;

};
```

Esempi utilizzo dati di tipo unione (2/2)

```
struct OggettoGrafico
{
    // è maggiore di zero se
    // l'oggetto grafico è un rettangolo
    int flagRettangolo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
        struct Punto pt;
        struct Rettangolo rect;
    } oggetto;

};

void DisegnaRettangolo(struct Rettangolo rect);
void DisegnaPunto(struct Punto pt);

void DisegnaOggettoGrafico(struct OggettoGrafico obj)
{
    if(obj.flagRettangolo)
        DisegnaRettangolo(obj.oggetto.rect);
    else
        DisegnaPunto(obj.oggetto.pt);
}
```

Tipi definiti da utente: enumerazione

- Nel linguaggio C, un'enumerazione, è un insieme finito di valori interi (**costanti intere di enumerazione**) rappresentati da nomi simbolici (identificatori).
- Le enumerazioni sono un mezzo efficiente per associare valori interi costanti a dei nomi.

Definizione di un tipo enumerativo (1/3)

Sintassi

```
enum <nome tipo>
{
    <dichiarazione 1>,
    <dichiarazione 2>,
    ....
    <dichiarazione N>
};
```

<dichiarazione i> ($1 \leq i \leq N$): è

- o un identificatore valido,
- oppure un'espressione della forma <nome> = <costante> dove nome è un identificatore valido e <costante> è una costante intera (valore costante dell'identificatore).

Dichiarazione di identificatori (nomi simbolici per costanti intere) racchiuse tra parentesi graffe, terminante con un punto e virgola, e avente come intestazione la parola chiave **enum** seguita da un **identificatore di tipo**.

Definizione di un tipo enumerativo (2/3)

Sintassi

```
enum <nome tipo>
{
    <dichiarazione 1>,
    <dichiarazione 2>,
    ....
    <dichiarazione N>
};
```

- Gli identificatori nelle varie dichiarazioni devono essere distinti e devono essere unici all'interno del programma.
- Ad identificatori distinti può essere associata la stessa costante intera.
- Se la prima dichiarazione non ha un valore esplicito, al primo identificatore viene assegnato un valore 0.
- Per una dichiarazione distinta dalla prima il cui valore non è esplicitamente assegnato, il valore del nome è ottenuto incrementando di 1 il valore del nome precedente.

Definizione di un tipo enumerativo (3/3)

Esempi:

```
//Assegna il nome FALSE al valore 0  
// e il nome TRUE al valore 1  
enum Bool { FALSE, TRUE };
```

```
//Mesi dell'anno: gli identificatori sono impostati,  
//rispettivamente con gli interi da 1 a 12  
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG, AGO, SETT, OTT, NOV,DIC};
```

- Per il campo d'azione o scope di una definizione di un tipo enumerativo valgono le stesse regole viste per le definizioni di tipo struttura.
- Per consentire l'accesso in più file sorgenti di una definizione di tipo, conviene porre la definizione in un file header ed includere il file tramite la direttiva **#include** del preprocessore.

Dichiarazione variabili di tipo enumerativo (1/2)

Sintassi senza inizializzazione:

enum <nome tipo> <var 1>, ..., <var N>;

Sintassi con inizializzazione:

enum <nome tipo> <var 1> = <id 1>, ..., <var N> =<id N>;

- **<nome tipo >**: identificatore della definizione di tipo enumerativo.
- **<var 1>, ..., <var N>**: nomi di variabili (qualsiasi identificatore valido).
- **<id 1>, ..., <id N>**: nomi simbolici per le costanti enumerative utilizzate nella definizione di tipo.

Esempio:

```
//Mesi dell'anno: gli identificatori sono impostati,  
//rispettivamente con gli interi da 1 a 12  
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG, AGO, SETT, OTT, NOV,DIC};  
  
enum Mesi meseNascita = FEB;  
enum Mesi mese1, mese2;
```

Dichiarazione variabili di tipo enumerativo (2/2)

Variabili di tipo enumerativo (come per i tipi struttura e unione) possono essere anche dichiarate immediatamente dopo la definizione dell'associato tipo enumerativo e prima del terminatore punto e virgola.

```
//Mesi dell'anno: gli identificatori sono impostati,  
//rispettivamente con gli interi da 1 a 12  
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG,  
           AGO, SETT, OTT, NOV,DIC} mese1 =GIU, mese2 = LUG;
```

Variabili di tipo enumerativo sono variabili numeriche e dunque possono essere utilizzate per comporre espressioni numeriche.

Esempi utilizzo dati di tipo enumerativo

```
enum TipoGrafico{RETTOANGOLO, PUNTO};
struct OggettoGrafico
{
    // tipo dell'oggetto grafico
    enum TipoGrafico tipo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
        struct Punto pt;
        struct Rettangolo rect;
    } oggetto;

};

void DisegnaRettangolo(struct Rettangolo rect);
void DisegnaPunto(struct Punto pt);

void DisegnaOggettoGrafico(struct OggettoGrafico obj)
{
    switch(obj.tipo)
    {
        case RETTANGOLO:
            DisegnaRettangolo(obj.oggetto.rect);
            break;
        case PUNTO:
            DisegnaPunto(obj.oggetto.pt);
    }
}
```

Tipi definiti da utente: array (1/2)

- Nel linguaggio C, un **array**, detto anche **vettore**, è una sequenza ordinata di dati dello stesso tipo, raggruppati sotto un unico nome.
- I singoli dati in un array sono chiamati **elementi dell'array** e possono essere acceduti tramite l'**operatore di indicizzazione** [].
- Elementi consecutivi in un array hanno **locazioni di memoria contigue**.
- Il numero di elementi di un array è chiamato **lunghezza dell'array**. Gli array sono in generale dati **a lunghezza variabile** dal momento che la lunghezza di un array può essere specificata tramite espressioni numeriche intere il cui valore è noto a tempo di esecuzione.

Tipi definiti da utente: array (2/2)

- A differenza degli aggregati (strutture e unioni) e dei tipi enumerativi, per definire una variabile di tipo array non bisogna prima definire un tipo array.
- Nella dichiarazione di una variabile di tipo array si specifica il nome della variabile, il tipo comune degli elementi di un array, e la lunghezza dell'array.
- **IMPORTANTE:** a differenza delle variabili semplici (numeriche) e delle variabili di tipo aggregato (struttura o unione) o di tipo enumerativo, una variabile array è una **variabile puntatore** dal momento che il suo valore rappresenta l'indirizzo di memoria della locazione di memoria associata al primo elemento dell'array.

Dichiarazione variabili array (1/2)

Sintassi per array monodimensionali:

<tipo elemento> <nome array> [<lunghezza array>];

- <tipo elemento>: rappresenta il tipo comune degli elementi. Per elementi semplici è l'insieme di parole chiave per identificare il tipo semplice (int, double, char, long double, ecc). Per i tipi struttura (risp., unione, risp., enumerazione) è la parola chiave **struct** (risp., **union**, risp., **enum**) seguita dal nome del tipo (come specificato nella definizione di tipo).
- <nome array>: nome dell'array (qualsiasi identificatore valido).
- <lunghezza array>: rappresenta la lunghezza dell'array. Può essere una qualsiasi espressione numerica di tipo intero.

Esempio: array di strutture

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};

//Array di nome traiettoria consistente di 10
//elementi di tipo Punto
struct Punto traiettoria[10];
```

Dichiarazione variabili array (2/2)

Sintassi per array monodimensionali:

<tipo elemento> <nome array> [<lunghezza array>];

- Per dichiarazioni di (variabili) array globali o locali statiche (e, cioè, il cui campo di memoria è statico), la lunghezza dell'array <lunghezza array> deve essere una espressione numerica intera costante il cui valore, determinabile a tempo di compilazione, deve essere un intero positivo. Altrimenti, il compilatore segnala un errore di sintassi. Per tali variabili, l'area di memoria viene allocata all'inizio dell'esecuzione del programma.
- Per dichiarazioni di variabili array locali automatiche, è possibile utilizzare espressioni numeriche intere generiche per specificare la lunghezza di un array. La valutazione a tempo di esecuzione dell'espressione numerica per la lunghezza dell'array (e la conseguente allocazione dinamica di un'area di memoria per memorizzare l'array) può generare un errore irreversibile se il valore ottenuto non è positivo e eccede la capacità di memoria a disposizione.

Inizializzazione variabili array (1/3)

Dichiarazione array monodimensionale con inizializzazione:

<tipo elemento> <nome array> [<lunghezza array>] =

<inizializzazione>;

<lunghezza array> deve essere un'espressione intera costante.

Sintassi <inizializzazione>: rappresenta la parte di inizializzazione.

{ <Init 1>,, <Init N> }

- N deve essere non superiore alla lunghezza dell'array.
 - **<Init i>** inizializza l' i -esimo membro ($1 \leq i \leq N$): per elementi semplici, è un'espressione numerica generica. Per elementi aggregati o di tipo enumerativo, è a sua volta un'espressione di inizializzazione oppure il nome di una variabile dello stesso tipo, oppure una chiamata a funzione restituente un dato di quel tipo.

Inizializzazione variabili array (2/3)

Dichiarazione array monodimensionale con inizializzazione:

<tipo elemento> <nome array> [<lunghezza array>] =

<inizializzazione>;

<lunghezza array> deve essere un'espressione intera costante.

Sintassi <inizializzazione>: rappresenta la parte di inizializzazione.

{ <Init 1>,,<Init N> }

- Se N è inferiore al numero di elementi dell'array, i restanti elementi sono inizializzati a zero (per elementi di tipo numerico).
 - Per **dichiarazioni di variabili globali o statiche**, le espressioni numeriche utilizzate devono essere costanti (i valori di inizializzazione devono essere noti a tempo di compilazione).

Inizializzazione variabili array (3/3)

```
struct Punto
{
    float x; //coordinata Lungo l'asse X
    float y; //coordinata Lungo l'asse Y
};

int cifre[10] ={0,1,2,3,4,5,6,7,8,9};

//Per il primo punto della traiettoria,
//l'ascissa x è inizializzata a 1.6 e
//e l'ordinata a 7.8. Per tutti gli
//altri punti, l'ascissa e l'ordinata sono
//inizializzati a zero.
struct Punto traiettoria[11] = {{1.6,7.8}};
```

Accesso agli elementi di un array (1/4)

- Ad ogni elemento di un array è associato un numero di posizione (**indice**) che va da 0 alla lunghezza dell'array meno 1.
- Si può far riferimento a uno qualunque degli elementi di un array monodimensionale fornendo il nome dell'array seguito dal numero di posizione (indice) dell'elemento racchiuso tra parentesi quadre.

Accesso agli elementi di un array (2/4)

Sintassi accesso agli elementi di un array monodimensionale.

<nome array>[<indice elemento>]

- **<indice elemento>** è una generica espressione numerica intera.
- L'espressione **<nome array>[<indice elemento>]** può essere utilizzata alla stessa stregua di una variabile avente come tipo il tipo degli elementi di un array. In particolare, può comparire come operando sinistro in un'istruzione di assegnazione e come parte di un'espressione numerica se il tipo degli elementi è numerico.
- Le parentesi quadre usate per racchiudere l'indice di un array sono considerate in C un operatore. L'**operatore di indicizzazione** [] ha la stessa precedenza dell'**operatore** () usato nelle chiamate di funzione (e, cioè, le parentesi tonde poste dopo il nome della funzione).

```
int A[10];
int i=3;
A[i] = i*i;
```

Accesso agli elementi di un array (3/4)

- È importante assicurarsi che ogni indice che si usa per accedere a un elemento di un array sia dentro i confini dell'array (e, cioè vada da 0 alla lunghezza dell'array meno 1).
- Il C *non fornisce alcun controllo automatico dei confini per gli array* (ciò è compito del programmatore).
- Consentire ai programmi di leggere e scrivere negli elementi di un array al di fuori dei confini dell' array rappresenta un comune difetto di sicurezza.
 - Leggere al di fuori dei confini può fare arrestare il programma o perfino dare l'impressione che questo venga eseguito correttamente, mentre invece usa dati non validi.
 - Scrivere al di fuori dei confini (**overflow del buffer**) può corrompere i dati in memoria di un programma e arrestare il programma.

Accesso agli elementi di un array (4/4)

Inizializzazione degli elementi di un array monodimensionale tramite ciclo controllato da contatore.

Esempio:

```
void Prova(struct Punto pt, int size)
{
    if(size >0)
    {
        struct Punto A[size];
        for(int i=0;i<size;i++)
        {
            A[i] = pt;
            pt.x++;
            pt.y++;
            printf("Punto %d ha ordinata %f\n",i,A[i].y);
        }
    }

int main ()
{
    struct Punto pt = {2.4,5.8};
    Prova(pt,10);
}
```

Precedenza e associatività operatore []

Operatori		Associatività	Tipo
[]	() ++ (postfisso) -- (postfisso)	da sinistra a destra	precedenza più alta
+ -	! ++ (prefisso) -- (prefisso) (tipo)	da destra a sinistra	unario
* / %		da sinistra a destra	moltiplicativo
+ -		da sinistra a destra	additivo
< <= > >=		da sinistra a destra	relazionale
== !=		da sinistra a destra	di uguaglianza
&&		da sinistra a destra	AND logico
		da sinistra a destra	OR logico
? :		da destra a sinistra	condizionale
= += -= *= /= %=		da destra a sinistra	di assegnazione
,		da sinistra a destra	virgola

Array monodimensionali e funzioni (1/4)

- Le funzioni possono avere come parametri formali variabili di qualsiasi tipo e, dunque, anche variabili array.
- In una funzione, la dichiarazione di un parametro formale di tipo array corrisponde alla dichiarazione di una variabile array ma la lunghezza non viene specificata.

Sintassi parametro formale array monodimensionale:

<tipo elemento> <nome parametro> []

- È possibile utilizzare un'altra notazione per specificare un parametro formale di tipo array (lo vedremo in seguito introducendo i puntatori).
- L'argomento (parametro attuale) in una chiamata di funzione deve essere il **nome di una variabile array** i cui elementi hanno lo stesso tipo degli elementi del parametro formale.

Array monodimensionali e funzioni (2/4)

- Una variabile array è una **variabile puntatore** (il suo valore è l'indirizzo di memoria del primo elemento dell'array).
- Quando in una chiamata di funzione viene passato come argomento una variabile array, viene copiato nel corrispondente parametro formale array l'indirizzo di memoria di partenza dell'array originario.
- Di conseguenza, quando la funzione chiamata modifica nel suo corpo gli elementi del parametro formale array, essa modifica gli effettivi elementi dell'array nelle loro originarie locazioni di memoria.
- In altri termini, la funzione chiamata può modificare i valori degli elementi nell'array specificato come argomento dalla funzione chiamante. Dunque, gli array **vengono passati per riferimento**.

Array monodimensionali e funzioni (3/4)

- Il passaggio degli array per riferimento ha senso per ragioni di prestazioni. Se gli array fossero passati per valore, verrebbe passata **una copia** di ogni elemento. Per array grandi, passati frequentemente, ciò comporterebbe un *overhead* considerevole sia in termini di tempo di computazione che di quantità di memoria.
- BUONA NORMA: in funzioni che prendono in input parametri array monodimensionali, utilizzare un altro parametro intero per specificare la lunghezza dell'array od il numero di elementi dell'array da leggere o modificare.

Array monodimensionali e funzioni (4/4)

Esempio:

```
#define SIZE 5

void ModificaArray(int b[],int size)
{
    if(size>0)
    {
        for(int j =0;j<size;j++)
            b[j] *= 2;
    }
}

int main ()
{
    int a[SIZE] ={0,1,2,3,4};

    //stampa l'array originario
    for(int i=0; i<SIZE;i++)
        printf("Elemento %d di array originario: %d\n",i,a[i]);

    ModificaArray(a,SIZE);

    //stampa l'array modificato
    for(int i=0; i<SIZE;i++)
        printf("Elemento %d di array modificato: %d\n",i,a[i]);
}
```

Output:

```
Elemento 0 di array originario: 0
Elemento 1 di array originario: 1
Elemento 2 di array originario: 2
Elemento 3 di array originario: 3
Elemento 4 di array originario: 4
Elemento 0 di array modificato: 0
Elemento 1 di array modificato: 2
Elemento 2 di array modificato: 4
Elemento 3 di array modificato: 6
Elemento 4 di array modificato: 8
```

Esempio utilizzo array monodimensionali

Calcolo somma degli elementi di un array di tipo numerico.

```
int CalcolaSomma(int b[], int size)
{
    int res = 0;
    if(size>0)
    {
        for(int j =0;j<size;j++)
            res += b[j];
    }
    return res;
}
```

Lezione 21

Antonio Origlia
a.a. 2022/2023

Sommario - Lezione 21: Esercizi e direttive del preprocessore

- Esercizi sulla ricorsione.
- Esercizi sulle liste.
- Direttive del preprocessore.
- Direttiva #define: costanti simboliche e macro.
- Compilazione condizionale.
- Typedef.

Esercizi sulla ricorsione (1/3)

1. **Ricorsione su stringhe:** scrivere una funzione ricorsiva che prende in input una stringa (array di caratteri terminante con '\0') ed eventualmente altri parametri (utili per la ricorsione) e stampi a video tutti gli anagrammi (permutazioni) della stringa data. Testare la funzione nel main chiamando la funzione con una stringa inserita da tastiera.

Suggerimento: la funzione ricorsiva prende in input come secondo parametro un puntatore ad uno dei caratteri non terminali della stringa (quando la funzione è chiamata per la prima volta, tale parametro punta al primo carattere della stringa). Questo puntatore individua la sottostringa che va dal carattere puntato fino alla fine della stringa data. È possibile utilizzare la funzione **strlen** per calcolare la lunghezza della sottostringa.

Esercizi sulla ricorsione (2/3)

Per esercitarsi sulla ricorsione definiamo una classe di espressioni aritmetiche (non sono nel linguaggio C) che utilizzano quattro operatori aritmetici utilizzati in notazione prefissa: + (somma), - (sottrazione), * (prodotto), divisione (/). Formalmente, un'espressione nel nostro linguaggio è definita induttivamente come segue:

- **Caso base:** una cifra decimale diversa da zero è un'espressione.
- **Caso induttivo:** se `<exp 1>` e `<exp 2>` sono espressioni nel nostro linguaggio, allora:
 - `+(<exp 1>, <exp 2>)` è un'espressione,
 - `-(<exp 1>, <exp 2>)` è un'espressione,
 - `/(<exp 1>, <exp 2>)` è un'espressione,
 - `*(<exp 1>, <exp 2>)` è un'espressione.
- Null'altro è un'espressione nel nostro linguaggio.

Esercizi sulla ricorsione (3/3)

Esempi di espressioni valide:

+ $(3,/(6,2))$ il cui valore è 6

2. **Interprete di espressioni:** scrivere una funzione ricorsiva che prende in input una stringa di caratteri (array di caratteri terminante con '\0'), un puntatore ad un carattere nella stringa, e :

- nel caso in cui esista una sottostringa a partire dal carattere dato che rappresenta un'espressione valida, la funzione deve restituire un puntatore all'ultimo carattere di tale sottostringa e come ulteriore parametro d'output di tipo **int** il valore denotato dall'espressione. Altrimenti, il puntatore restituito deve essere NULL.

Utilizzare una funzione ausiliaria che data una cifra decimale restituisca il valore intero associato. Testare la funzione ricorsiva nel main chiamando la funzione con una stringa inserita da tastiera.

Ad esempio, per la stringa “g+(3,4)f” ed un puntatore al secondo carattere, la funzione restituirà un puntatore al carattere ‘)’ e come parametro d'output l'intero 7. Per la stringa +(3,/(6,2)), la funzione restituirà il puntatore all'ultimo carattere e come parametro d'output il valore 6 dell'espressione.

Esercizi sulle liste (1/2)

3. Definire una funzione che prenda in input i puntatori ai nodi iniziali di due liste concatenate (si assume nel seguito che il campo dati di un nodo di una lista sia un intero), fonda le due liste in un'unica lista in cui i nodi con valore pari precedano i nodi con valore dispari, e restituisca un puntatore al nodo iniziale di tale lista. Testare la funzione utilizzando funzioni ausiliare di allocazione e inizializzazione di una lista con dati inseriti da tastiera.
4. Definire una funzione che prenda in input i puntatori ai nodi iniziali di due liste L1 ed L2 e restituisca un puntatore al nodo iniziale di una lista senza ripetizioni (due nodi distinti devono avere valori distinti) costruita da L1 ed L2 e che contiene tutti e solo gli elementi di L2 non appartenenti a L1. Testare la funzione come nell'esercizio 3.

Esercizi sulle liste (2/2)

5. Definire una funzione che prenda in input il puntatore al nodo iniziale di una lista concatenata L ed un intero $k \geq 1$ e restituisca un puntatore al nodo iniziale di una nuova lista senza ripetizioni che contenga tutti e solo gli elementi di L che si ripetono in L esattamente k volte.

Direttive del preprocessore

Le direttive del preprocessore consentono di eseguire azioni di inclusione e sostituzione di testo prima che il programma venga compilato. Tutte le direttive iniziano con il carattere # e le più importanti sono:

- Direttive **#include** per l'inclusione di file header nel file sorgente da compilare.
- Direttive **#define** per la definizione di costanti simboliche e macro.
- Direttive condizionali per la **compilazione condizionale** del codice del programma.

Direttiva #define senza argomenti (1/3)

La direttiva **#define** senza argomenti consente di definire *costanti simboliche* (costanti rappresentate come simboli) e *macro* senza argomenti (operazioni rappresentate come simboli).

Sintassi:

#define <identificatore> <testo>

- **<identificatore>** è un qualsiasi identificatore valido.
- **<testo>** (testo di sostituzione) è un testo arbitrario: se il testo è più lungo di una riga, si deve mettere il carattere di continuazione **backslash** (\) alla fine di ogni riga (ad eccezione dell'ultima), ad indicare che il testo di sostituzione continua sulla riga successiva.
- Il testo di sostituzione può contenere anche identificatori definiti tramite precedenti direttive **#define**.

Direttiva #define senza argomenti (2/3)

Sintassi:

#define <identificatore> <testo>

Semantica:

Tutte le occorrenze successive dell'*identificatore* che non appaiono in *stringhe letterali* (sequenze di caratteri racchiuse tra doppi apici) o *commenti* vengono rimpiazzate automaticamente con il testo di sostituzione **<testo>** prima della compilazione del programma.

Campo d'azione:

Il campo di azione (scope) di un identificatore (nome) definito con una **#define** va dal punto in cui la **#define** si trova fino alla fine del file sorgente.

Direttiva #define senza argomenti (3/3)

Esempio:

```
#define PI 3.14159  
#define MESSAGGIO "Benvenuto utente!"
```

- L'utilizzo di costanti simboliche fa sì che i programmi siano più facili da modificare. Piuttosto che cercare ogni occorrenza di un valore nel codice, basta modificare una costante simbolica una volta nella sua direttiva **#define**. Quando il programma viene ricompilato, tutte le occorrenze di quella costante nel programma vengono modificate di conseguenza.
- Per costanti numeriche e costanti stringhe, un'alternativa valida è definire variabili globali con il qualificatore **const** come nel seguente esempio.

```
const float PI = 3.14159;  
const char Messaggio[] ="Benvenuto utente!";
```

Direttiva #define con argomenti (1/4)

La direttiva **#define** con argomenti viene utilizzata per definire **macro con argomenti** utilizzate di solito per sostituire chiamate di funzioni con un codice in linea, così da eliminare il sovraccarico delle chiamate di funzioni.

Sintassi:

#define <identificatore>(<arg 1>,...<arg N>) <testo>

- **<arg 1> .. <arg N>** (lista degli argomenti): sono identificatori validi distinti.

Direttiva #define con argomenti (2/4)

Sintassi:

#define <identificatore>(<arg 1>,...<arg N>) <testo>

Semantica:

Tutte le occorrenze successive di espressioni della forma

<identificatore>(<exp 1>,...<exp N>)

che non appaiono in *stringhe letterali* (sequenze di caratteri racchiuse tra doppi apici) o *commenti* vengono rimpiazzate automaticamente con il testo ottenuto da **<testo>** sostituendo ogni occorrenza di **<arg i>** in **<testo>** con **<exp i>** (**espansione della macro**).

Campo d'azione:

Lo stesso di una **#define** senza argomenti.

Direttiva #define senza argomenti (3/4)

Esempio:

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

Ogni invocazione della macro viene espansa in codice in linea, dove ogni occorrenza di un parametro formale (**A** e **B** nel nostro esempio) viene rimpiazzata con il corrispondente argomento reale. Ad esempio, l'istruzione:

```
x= max(p+q,r+s);
```

viene sostituita con

```
x = ((p+q) >(r+s) ? (p+q): (r+s))
```

Direttiva #define senza argomenti (4/4)

Esempio:

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

Dal momento che in questa macro un parametro formale occorre più volte, si possono avere dei problemi quando gli argomenti attuali sono espressioni con effetti collaterali (modifica del valore di una variabile). Ad esempio,

```
max(i++,j++); /*SBAGLIATO*/
```

incrementa due volte la variabile massima tra i e j. Inoltre, nelle macro con argomenti è necessario porre un'attenzione particolare sull'uso delle parentesi, in modo da essere certi che l'ordine delle valutazioni sia quello desiderato. Ad esempio, ciò accade quando la macro

```
#define square(x) x*x
```

viene invocata nella forma **square(z+1)**.

Direttiva #undef (1/4)

#undef <identificatore>

“rimuove la definizione” di un identificatore definito con una direttiva **#define** (costante simbolica o macro). Una volta rimosso, un nome può essere ridefinito con **#define**. Ad esempio,

```
#include "ProgettoEsterno.h"

#undef MACRO
#define MACRO
```

non conoscendo totalmente *ProgettoEsterno.h*, prima di definire una propria macro, conviene usare **#undef** per prevenire una doppia definizione.

Compilazione condizionale (1/3)

Il linguaggio C fornisce direttive di preprocessamento, chiamate **direttive condizionali**, che consentono la **compilazione condizionata**, vale a dire la compilazione di parte del codice sorgente solo sotto certe condizioni dove quest'ultime sono valutate durante il preprocessamento.

Le direttive condizionali sono utili per diversi scopi tra cui:

- Per evitare che i file di intestazione siano inclusi innumerevoli volte nello stesso file sorgente.
- Per la portabilità del codice: attivando/disattivando porzioni di codice che rendono il programma compatibile con una serie di piattaforme.
- Per il debug del codice sorgente.

Compilazione condizionale (2/3)

Una direttiva condizionale consiste di un comando che inizia con il carattere # seguito (dopo spazi bianchi) da una **condizione** che deve comparire sulla stessa riga del comando e può continuare su righe successive a patto di terminare ogni linea con il carattere \.

- La **condizione** della direttiva deve essere un'espressione numerica **costante di tipo intero** che non può contenere né occorrenze dell'operatore di *cast*, né occorrenze dell'operatore *sizeof*, né occorrenze di *costanti di enumerazione*.
- Ogni costante intera nella condizione viene considerata di tipo **long** in modo che tutte le operazioni aritmetiche si svolgano su interi di tipo **long** o **unsigned long**.

Compilazione condizionale (3/3)

Una condizione di una direttiva condizionale può contenere sotto-espressioni della forma:

defined <identificatore>

o equivalentemente

defined (<identificatore>)

Il preprocessore assegna all'espressione precedente il valore 1 se l'identificatore associato è stato già definito tramite una direttiva **#define** e tale definizione non è stata eliminata tramite una corrispondente direttiva **#undef**. Altrimenti, all'espressione viene assegnato il valore 0.

Direttive condizionali

- **#if <condizione>**: include nella compilazione un qualche testo in dipendenza del valore dell'espressione costante **<condizione>**.
- **#ifdef <identificatore>**: abbreviazione di
#if defined <identificatore>
- **#ifndef <identificatore>**: abbreviazione di
#if !(defined <identificatore>)
- **#endif**: ogni costrutto **#if** deve essere seguito nel testo da un corrispondente costrutto **#endif**.
- **#elif <condizione>**: un costrutto condizionale del preprocessore costituito da più parti può essere scritto usando le direttive **#elif** (l'equivalente di **else if** in un'istruzione **if**) e **#else** (l'equivalente di **else** in un'istruzione **if**).
- **#else**.

Direttiva condizionale semplice

SINTASSI

```
#if <condizione>
  <testo>
#endif
```

SEMANTICA

Se **<condizione>** assume un valore diverso da zero allora **<testo>** viene compilato. Altrimenti, **<testo>** non viene compilato.

<testo> può essere una generica porzione del programma che può contenere anche direttive del preprocessore ma non può contenere direttive **#elif** o **#else** che non siano precedute da una direttiva **#if**.

Direttiva condizionale doppia

SINTASSI

```
#if <condizione>
  <texto 1>
#else
  <texto 2>
#endif
```

SEMANTICA

Se **<condizione>** assume un valore diverso da zero allora viene compilato solo **<texto 1>**. Altrimenti, viene compilato solo **<texto 2>**.

<texto 1> e **<texto 2>** non possono contenere direttive **#elif** o **#else** che non siano precedute da una direttiva **#if**.

Direttiva condizionale a più rami

SINTASSI

```
#if <condizione 1>
  <testo 1>
#elif <condizione 2>
  <testo 2>
.....
#elif <condizione N>
  <testo N>
#else
  <testo>
#endif
```

SEMANTICA

Vengono valutate in ordine le condizioni <condizione 1> .. <condizione N> fino a quando la condizione correntemente esaminata assume un valore diverso da 0. In questo caso, viene compilato solo il testo associato alla corrispondente direttiva. Se invece tutte le condizioni assumono valore zero allora viene compilato solo il testo, associato alla direttiva #else.

- La parte **else** è opzionale.
- <testo 1> ... <testo N> e <testo> non possono contenere direttive #elif o #else che non siano precedute da una direttiva #if.

Esempio: codice di debugging

La compilazione condizionale si usa talvolta come aiuto per il debugging. Ad esempio, è possibile utilizzare istruzioni **printf** per stampare valori delle variabili. Queste istruzioni **printf** possono essere racchiuse in direttive condizionali per il preprocessore.

Esempio:

```
#define DEBUG 1
.....
#if DEBUG
printf("debug: a = %d\n",a);
#endif
```

Visto che DEBUG è diverso da 0, la linea del printf viene compilata ed aiuta nel debug del programma. Una volta terminata la fase di test del programma, si sostituisce 1 con 0 nella definizione di DEBUG in modo che le istruzioni printf per il debug non vengano più compilate.

Esempio: inclusione di file header

Per assicurarsi che il contenuto di un file header venga incluso una sola volta, possiamo racchiudere tale contenuto all'interno di una direttiva condizionale come la seguente:

```
#ifndef NOMEHEADER
#define NOMEHEADER

//Contenuto del file header nomeheader.h

#endif
```

Come identificatore nella `#define` conviene utilizzare il nome del file header (dove tutte le lettere sono convertite in maiuscolo) per garantire che la costante simbolica sia definita solo nello specifico file header.

Typedef

La parola chiave **typedef** fornisce un meccanismo per creare sinonimi (o alias) per tipi di dati precedentemente definiti.

- Ad i tipi struct sono spesso associati nomi di tipo più brevi tramite una **typedef**.

Esempio:

```
struct Punto
{
    float x; //coordinata Lungo L'asse X
    float y; //coordinata Lungo L'asse Y
};

//definisce un alias per il tipo struct Punto
typedef struct Punto PUNTO;

PUNTO pt;
pt.x=23;
```

Typedef per tipi predefiniti

Spesso si usa **typedef** per creare sinonimi di tipi primitivi. Ad esempio, un programma che richiede interi di quattro byte può usare il tipo **int** su un sistema e il tipo **long** su un altro. I programmi progettati per la portabilità usano spesso **typedef** per creare un alias per interi di quattro byte, ad esempio **Integer**. Il tipo effettivo associato ad **Integer** deve essere cambiato in un solo punto (la corrispondente **typedef**) per consentire il corretto funzionamento del programma su entrambi i sistemi.