INTRODUZIONE E FONDAMENTI

«**Programmazione in C**»

Kim N. King, Apogeo

Capitolo 1 e 2

Il Linguaggio C

Il Linguaggio di Programmazione C è costituito da

- un linguaggio di computazione (il C vero e proprio)
- un linguaggio di coordinazione fornito come "allegato", sotto forma di librerie standard:
  - alcune istruzioni "complesse" del linguaggio potrebbero essere realizzate in realtà da "mini-programmi" forniti insieme al compilatore, che li utilizza incorporandoli quando occorre.
  - LIBRERIE DI SISTEMA: insieme di componenti software che consentono di interfacciarsi col sistema operativo, usare le risorse da esso gestite, e realizzare alcune "istruzioni complesse" del linguaggio

Il **sistema operativo** è un programma di controllo che svolge operazioni fondamentali, risiede in una memoria interna permanente e interpreta i comandi utente che richiedono varie specie di servizi, come la visualizzazione, la stampa o la copia di un file, il raggruppamento logico dei file in una directory o l'esecuzione di un programma. In altre parole *il sistema operativo si occupa di gestire tutte le periferiche del calcolatore, tutti i processi, i dati di input/output.*

Origini

Fu inizialmente creato nei laboratori della AT&T Bell Laboratories (anni ’70) per lo sviluppo dei sistemi operativi da Ken Thompson, Dennis Ritchie ed altri.

UNIX scritto in assembly (programmi faticosi da gestire, migliorare, …)

Thompson creò un piccolo linguaggio (B) per un ulteriore sviluppo di UNIX (basato su BCPL, a sua volta basato su Algol 60) e riscrisse una porzione di UNIX in B.

A partire dal ‘71 Ritchie iniziò lo sviluppo di una versione stesa del linguaggio (NB, NewB) che diventò C, stabile dal 1973, tanto che UNIX venne riscritto in C.

Importante beneficio: la **portabiltà**, scrivendo compilatori C per altri computer psenti nei laboratori Bell, il team potè far funzionare UNIX anche su tutte quelle macchine.

Standardizzazione

Nel 1978 venne pubblicato il primo libro sul C "The C Programming Language", Kernigan, Ritchie, che divenne lo standard *de facto* (**K&R C**)

Nel 1980 C si era espanso ben oltre il mondo UNIX con compilatori disponibili su grande varietà di calcolatori con sistemi operativi differenti

Ma i programmatori che scrivevano nuovi compilatori si basavano su K&R, approssimativo su alcune caratteristiche del linguaggio

Inoltre il C continuò a cambiare anche dopo la pubblicazione del K&R

Necessità: descrizione del linguaggio precisa, accurata e aggiornata (**standard**) senza la quale i numerosi dialetti avrebbero minacciato la **portabilità** dei programmi C

Venne messo a punto uno standard statunitense chiamato ANSI C (American National Standards Institute) approvato nel 1989, poi approvato anche dell'ISO (International Organization for Standardization) nel 1990 (**C89 o C90**)

Nuovi cambiamenti vennero apportati nel 1999 (**C99**) con la pubblicazione del nuovo standard ISO

Standardizzazione

Il C99 non è ancora universalmente diffuso e ci vorranno anni affinchè tutti i compilatori diventino *C99-compliant*. Per un uso approfondito del linguaggio servirà conoscere le differenze fra C89 e C99. Ad esempio:

- **//commenti:** il C99 aggiunge secondo tipo di commenti
- **Identificatori**: C89 primi 31 caratteri significativi per gli identificatori, C99 63 caratteri
- **Valori restituiri dal main**: C89 funzione senza return valore restituito al s.o. indefinito, C99 se main dichiarato che restituisce int allora valore restituito al s.o. è 0
- **Dichiarazioni variabili**: C99 include la possibilità di dichiarare una variabile in un punto qualsiasi prima del suo utilizzo all'interno di un blocco (dichiarazioni e istruzioni anche mischiate)
- **Tipo restituito da funzione**: C99 non ammette l'omissione del tipo restituito da una funzione
- **Header <complex.h>**: C99 introduce <complex.h> che fornisce funzioni per eseguire operazioni matematiche sui numeri complessi
- …

Per un uso pienamente soddisfacente del C nell'implementazione di algoritmi le caratteristiche aggiunte al linguaggio con gli standard successivi non sono fondamentali

Linguaggi basati sul C

Enorme influenza sui linguaggi di programmazione moderni:

- C++: include tutte le caratteristiche del C e aggiunge il supporto alla programmazione orientata agli oggetti
- Java: basato sul C++ eredita molte delle caratteristiche del C
- C#: un più recente linguaggio basato su C++ e Java
- Perl: linguaggio di scripting che adotta molte delle caratteristiche del C

Linguaggio basilare per lo sviluppo di qualsiasi applicazione specialmente se memoria o potenza di calcolo sono limitate e cruciali, compresa l'implementazione di architetture operative; il Kernel di Linux è in gran parte scritto in tale linguaggio, di server web ed application server quali Apache, Tomcat, Websphere, IIS

Pregi

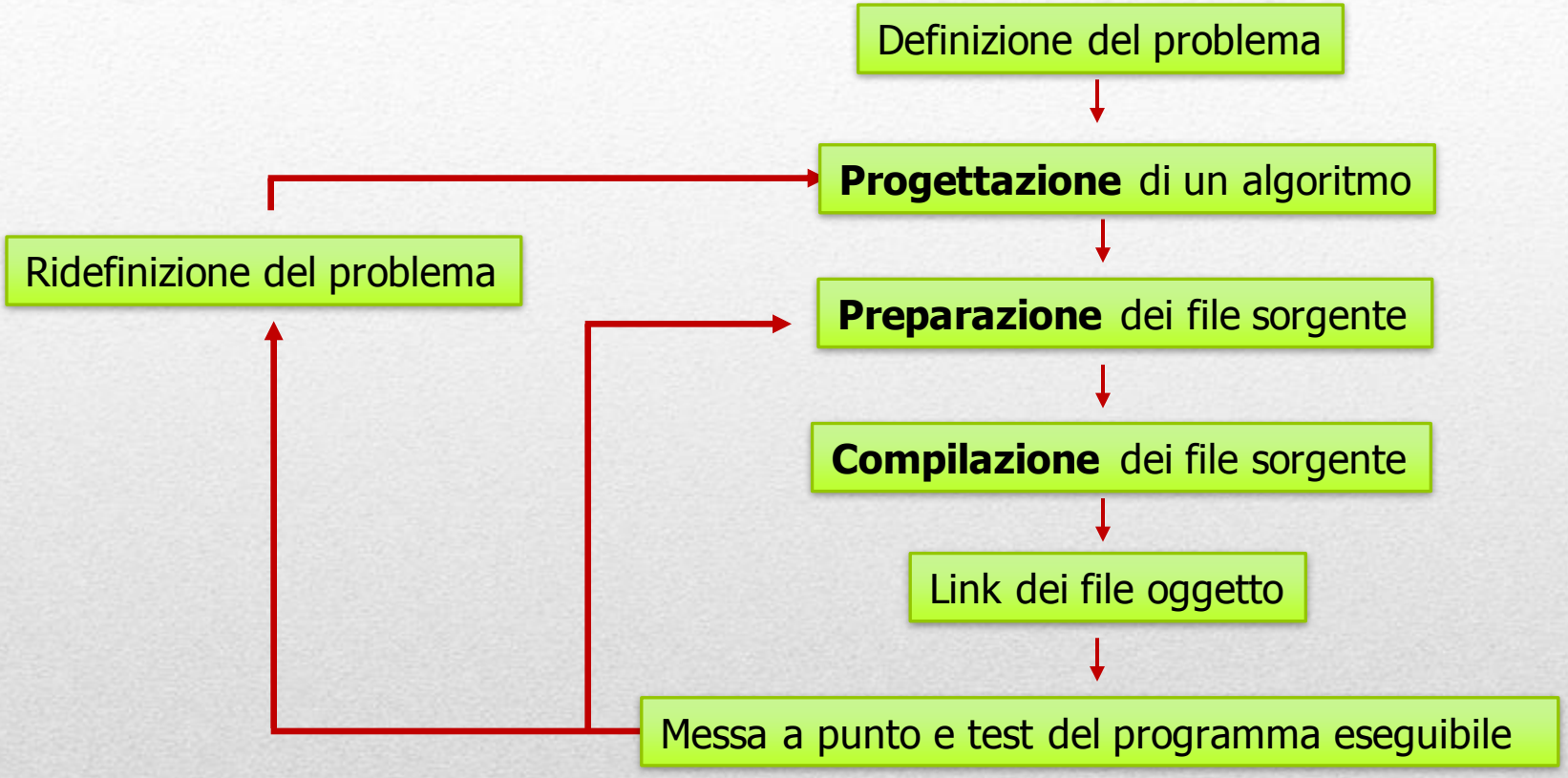
- **Efficienza:** pensato per applicazioni dove tradizionalmente veniva usato il linguaggio assembly era cruciale che i programmi in C potessero girare velocemente e con una quantità di memoria limitata; ad es. include la gestione dei **puntatori**, fornendo al programmatore una **visibilità molto elevata sulla memoria** della macchina
- **Tipizzato**: debole controllo sui tipi di dato; a differenza di altri linguaggi il C permette di operare con assegnamenti e confronti su dati di tipo diverso, in qualche caso solo mediante una conversione di tipo esplicita (cast). Il compilatore demanda al programmatore il compito di verificare la correttezza semantica delle espressioni e la gestione di eventuali errori generati da espressioni non corrette (**flessibilità**)
- **Basso livello**: più vicino al linguaggio macchina con istruzioni basilari che vengono elaborate direttamente dal processore e permettono un totale accesso alle risorse della macchina; produce quindi un **codice più compatto ed efficiente**
- **Portabilità**: Consente lo sviluppo di **programmi facilmente portabili** da una piattaforma ad un'altra; esiste una normativa ANSI che stabilisce le caratteristiche standard di un compilatore C e la modalità standard di programmazione in C
- Disponibilità di **librerie standard** che contengono centinaia di funzioni deputate all'i/o, alla manipolazione delle stringhe, alla gestione della memorizzazione, …

Debolezze

- **Programmi inclini agli errori** grazie alla flessibilità del C e alle possibili difficoltà per programmatori non accorti (provate a dimenticare un «&» ove necessario!)
- **Programmi difficili da capire** a causa della natura stringata e succinta dei programmi (linguaggio conciso) e della sua flessibilità se esasperata
- **Programmi difficili da modificare** se non sono stati sviluppati considerando la necessità di manutenzione del codice

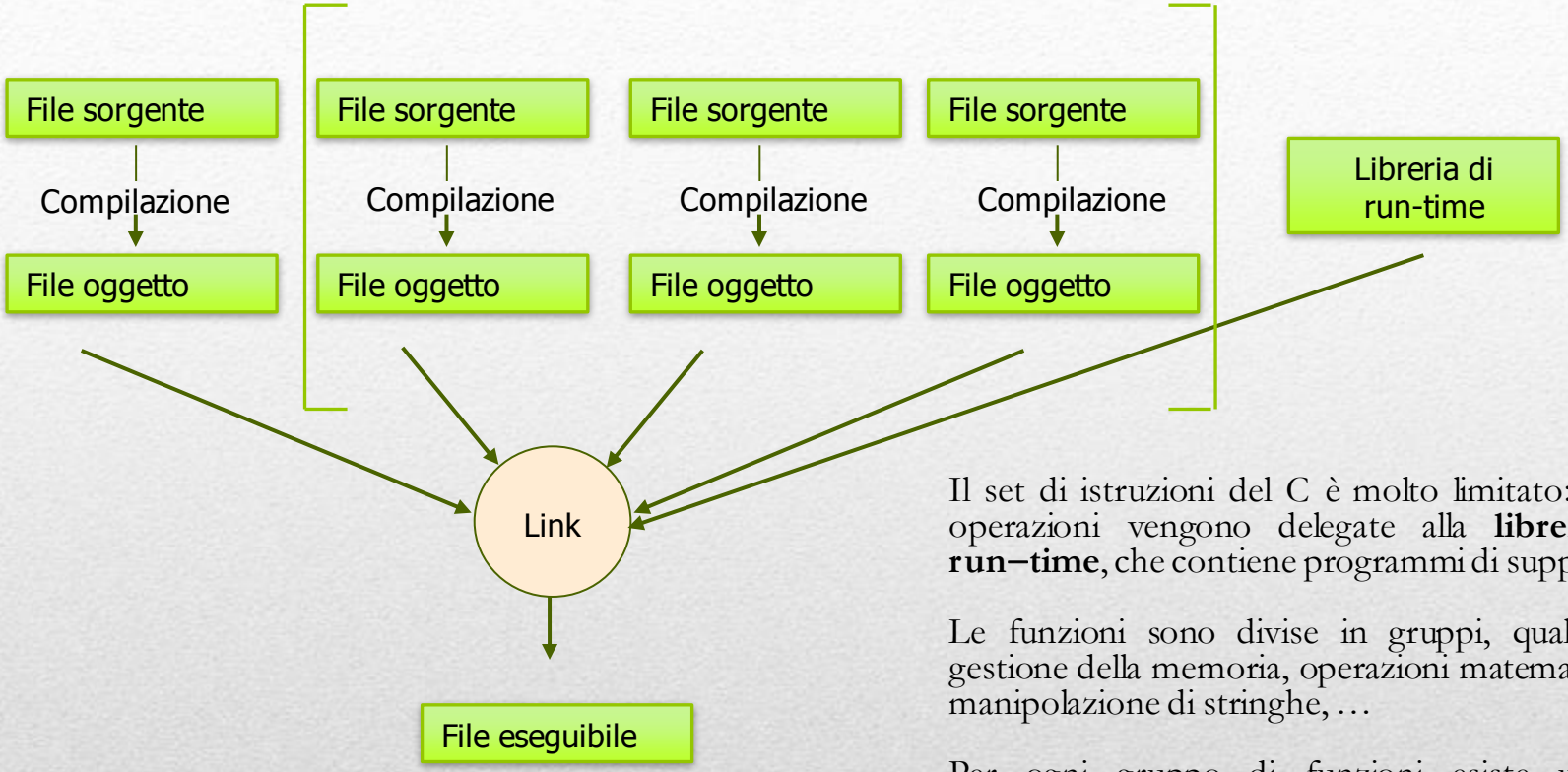
The International Obfuscated C Code Contest http://www.ioccc.org/

Lo sviluppo dei programmi



N.B.: Gli argomenti di inizio corso saranno tutti approfonditi nel seguito; ora abbiamo bisogno di porre le fondamenta dei diversi temi e di cominciare a lavorare sulla macchina

Lo sviluppo dei programmi



Il set di istruzioni del C è molto limitato: molte operazioni vengono delegate alla **libreria di run–time**, che contiene programmi di supporto

Le funzioni sono divise in gruppi, quali I/O, gestione della memoria, operazioni matematiche e manipolazione di stringhe, …

Per ogni gruppo di funzioni esiste un file sorgente, chiamato ***file header*,** contenente le informazioni necessarie per utilizzare le funzioni

I codici sorgente ed oggetto possono essere suddivisi in più file, il codice eseguibile di un programma risiede in un unico file

Un semplice programma in C: scrittura di una riga di testo

```
1
2  /*   A first program in C */
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```

IL PROGRAMMA SORGENTE

```
Welcome to C!
```

IL RISULTATO

- Commenti
  - Il testo comprteso fra /* e */ è ignorato dall'elaboratore
  - E' utile per commentare e descrivere il programma
- #include <stdio.h>
  - Direttiva del preprocessore: dice all'elaboratore di caricare la libreria <stdio.h> che contiene le operazioni di input/output standard poichè il C non ha comandi incorporati di lettura/scrittura

Commenti al programma

- int main()
  - Una delle funzioni in tutti i programmi in C deve essere main
  - Le parentesi tonde indicano una funzione
  - int significa che main "restituisce" un valore di tipo intero. Il valore restituito dalla funzione main (tipicamente un intero) può essere utilizzato da un eventuale programma chiamante.
  - Fino a quando non impareremo a scrivere altre funzioni, il *main* sarà l'unica dei nostri programmi
- Le parentesi graffe indicano un blocco di programma
  - Il "corpo" delle funzioni deve essere racchiuso fra parentesi graffe

Funzione: dalla matematica (regola per calcolare un valore a partire da uno o più argomenti dati) indica un raggruppamento di istruzioni al quale è stato assegnato un nome. Alcune calcolano un valore, altre no. Nel primo caso usiamo l'istruzione *return* per specificare il valore che restituisce

Commenti al programma

- printf("Welcome to C!\n");
  - Indica l'esecuzione di un'azione
    - Stampa la stringa di caratteri compresa fra gli apici
  - Tutta la linea è detta istruzione
    - Tutte le istruzioni devono terminare con il ;
  - \ - carattere di escape
    - Indica che printf deve eseguire qualcosa di particolare
    - \n è il carattere di "a capo"
- return 0;
  - E' un modo di "uscire" da una funzione
  - return 0, in questo caso indica che il programma termina normalmente
- Parentesi graffa }
  - Indica che si è raggiunta la fine del main

Altri esempi

```
/*
 * File esempio1.c
 * Scopo: utilizzare istruzione printf
 * Autore: Antonella Carbonaro
 * Data: 22 settembre 2014
*/

#include <stdio.h>
int main( )
{
   printf("Welcome to C!\n");
   return 0;
}
```

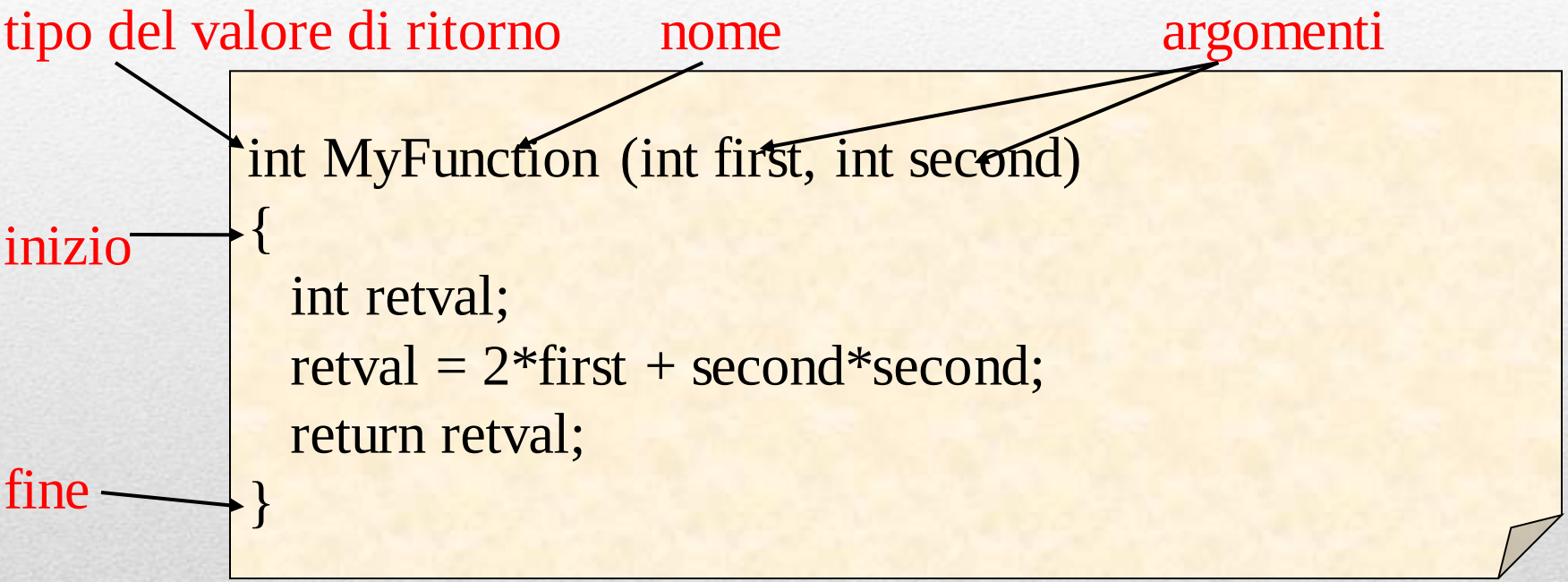
```
// File esempio2.c
// Scopo: stampare su un'unica riga
// con due istruzioni printf
// Autore: Antonella Carbonaro
// Data: 22 settembre 2014

#include <stdio.h>
int main( )
{
   printf("Welcome ");
   printf("to C!\n");
   return 0;
}
```

Caratteristiche di ogni programma C

- Ogni programma C deve contenere una ed una sola funzione main(), che rappresenta il programma principale, ed un punto di inizio dell'esecuzione del programma.
- La parentesi graffa aperta { indica l'inizio di un blocco di istruzioni
- La parentesi graffa chiusa } indica la fine di un blocco di istruzioni
- Per ogni { deve essercene una chiusa }
- I commenti possono essere posti ovunque e verranno ignorati dal compilatore
- /* questo è l'inizio del commento e questa la sua fine */
- Il C99 prevede un secondo tipo di commenti che iniziano con // e che terminano automaticamente alla fine della riga
- L'annidamento di commenti non è permesso
- Il C è case sensitive.

Generalizziamo

Unità fondamentale: ***funzione***



main: nome speciale di funzione dalla quale inizia l'esecuzione del programma

L'aspetto di un programma C

Tutti i programmi C sono costituiti da **almeno una funzione**: al crescere della complessità dei programmi e della capacità di organizzare un programma in sottoprogrammi, tale numero aumenterà.

La funzione che è sempre presente in tutti i programmi è chiamata **main**. Più in generale ci si riferisce alla funzione main come al "programma principale", in quanto è la prima funzione che viene chiamata quando si esegue un programma.

Il linguaggio C è costituito da sole **32 parole chiave riservate** che combinate realizzano la sintassi formale del linguaggio. Si noti che tutte le parole chiave vengono scritte in minuscolo. Una parola chiave non può essere utilizzata per altri scopi, quali ad esempio il nome di una variabile.

Il linguaggio di programmazione C può essere utilizzato per un **approccio strutturato** alla programmazione

La struttura di un programma C

```
direttive per il preprocessore
dichiarazioni globali
main( )
{
  variabili locali alla funzione main ;
  istruzioni della funzione main
}
f1( )
{
  variabili locali alla funzione f1 ;
  istruzioni della funzione f1
}
f2( )
{
  variabili locali alla funzione f2 ;
  istruzioni della funzione f2
}
 . . .
 etc
```

Anche il più semplice programma C si basa su tre **componenti chiave** del linguaggio:

- le **direttive**: modificano il programma prima della compilazione; iniziano con il carattere # e non terminano con il ;
- le **funzioni**: ad es. il *main*
- le **istruzioni**: comandi eseguiti durante l'esecuzione del programma

Si noti l'utilizzo delle parentesi tonde e graffe.

Le parentesi tonde ( ) vengono utilizzate in unione con i nomi delle funzioni, mentre le parentesi graffe {} vengono utilizzate per delimitare le espressioni.

Infine il punto e virgola ; indica la terminazione di un'espressione in C.

Il linguaggio C ha un formato piuttosto flessibile: espressioni lunghe possono essere continuate su righe successive senza problemi: il punto e virgola segnala al compilatore che è stata raggiunta la fine dell'espressione.

Inoltre, è possibile introdurre spazi indiscriminatamente, in genere allo scopo di dare un miglior aspetto al programma.

Un errore molto comune è dimenticarsi il punto e virgola: in questo caso il compilatore concatenerà più linee di codice generando un'espressione priva di qualsiasi significato. Per questo motivo il messaggio d'errore che il compilatore restituisce non è la mancanza del punto e virgola, quanto la presenza di un qualcosa di incomprensibile. Attenzione dunque a non scordarsi i punti e virgola e ad interpretare i messaggi del compilatore.

Le direttive al preprocessore

Una direttiva al preprocessore *(#include)* permette di inserire tutto il testo del file specificato nel nostro sorgente, a partire dalla riga in cui si trova la direttiva stessa.

In particolare, in *stdio.h* è descritto il modo in cui la funzione printf() si interfaccia al programma che la utilizza; ci sono poi altre direttive al preprocessore e definizioni che servono al compilatore per tradurre correttamente il programma.

Ogni compilatore C è accompagnato da un certo numero di file *.h*, detti **include file** o **header file**, il cui contenuto è necessario per un corretto utilizzo delle funzioni di libreria (anche le librerie sono fornite col compilatore).

Gli header contengono i "**prototipi**" (le dichiarazioni) delle funzioni di libreria e la definizione di costanti. La parola chiave "*include*" non è una istruzione del linguaggio C e quindi non deve essere terminata da *;*. Esempio:

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

Le librerie

Contengono codice che esegue funzioni quali ad es. le funzioni matematiche, le funzioni di I/O, le funzioni grafiche, …

In pratica tutte le **operazioni di interazione** tra i programmi e l'hardware, il firmware ed il sistema operativo sono delegate a funzioni (aventi interfaccia più o meno standardizzata) esterne al compilatore, il quale non deve dunque implementare particolari capacità di generazione di codice, peculiari per il sistema al quale è destinato, permettendo la sua **portabilità**.

Molti compilatori mettono a disposizione librerie proprietarie, che facilitano la programmazione ma che rendono difficile il porting del programma da un'architettura ad una differente.

I cosiddetti "ambienti di sviluppo integrati", quali ad es. Microsoft Visual C, generalmente facilitano la soluzione delle problematiche relative alle librerie da usare, così come nascondono la maggior parte degli aspetti della compilazione, mediante un'interfaccia visuale, e facilitano le fasi di debugging, permettendo di seguire passo passo l'esecuzione del programma.

Le librerie standard

Per usare una libreria, non occorre inserirla esplicitamente nel progetto: ogni ambiente di sviluppo sa già dove cercarle

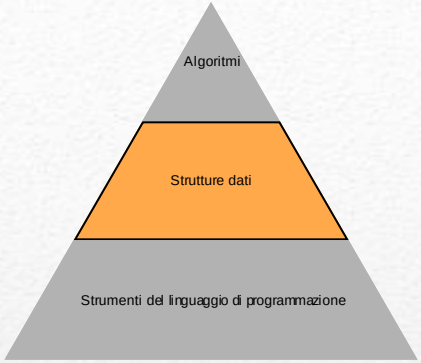
Ogni file sorgente che ne faccia uso deve però includere lo header opportuno, che contiene le dichiarazioni necessarie.

•input/output	stdio.h
•funzioni matematiche	math.h
•gestione di stringhe	string.h
•operazioni su caratteri	ctype.h
•gestione dinamica della memoria	stdlib.h
•operazioni su data e ora	time.h
•…	
•... e molte altre.	

RISOLUZIONE DI UN PROBLEMA

«Algoritmi e strutture dati»
Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano, McGraw Hill
Capitolo 3

Risoluzione di un problema



La risoluzione di un problema mediante la realizzazione di un programma richiede i seguenti passi:

1. Definizione di un modello che sintetizzi le caratteristiche essenziali del problema (l'algoritmo di soluzione a questo stadio può essere espresso solo in termini informali)
2. Definizione dell'**algoritmo** in uno pseudo-linguaggio, tramite **affinamenti successivi** fino a quando non vengono specificate in un certo dettaglio le operazioni che vanno compiute sui tipi di dati. Vengono cioè creati dei tipi di dato astratti (**abstract data type**) in cui ciascuna operazione sul tipo è una funzione con nome appropriato.
3. **Realizzazione dell'ADT** e delle **funzioni** che su di esso operano e delle restanti parti di algoritmo.

Risoluzione di un problema

Un mattone essenziale nella realizzazione di programmi è la procedura (funzione) che ha le seguenti caratteristiche:

- generalizza la nozione di operatore consentendo l'estensione di quelli forniti dal linguaggio (es. moltiplicazione di matrici)
- incapsula parti di un algoritmo localizzando in una sezione del programma tutte le istruzioni riguardanti aspetti comuni.

Risoluzione di un problema

Un ADT è invece un modello matematico insieme con la collezione delle operazioni definite su quel modello. Inoltre:

- generalizza i tipi primitivi (reale, intero, …) come una procedura fa con gli operatori sui tipi primitivi
- incapsula un tipo e le operazioni valide su di esso come una procedura fa con una parte di programma. Così i cambiamenti ad un ADT hanno impatto solo all'interno dell'ADT, l'interfaccia verso l'esterno resta identica e i dettagli non necessari sono invisibili dall'esterno.

Specifica *cosa* un'operazione (funzione) che agisce sul tipo di dato deve fare, ma non *come* l'operazione (funzione) può essere realizzata e soprattutto *come* gli oggetti della collezione possono essere organizzati in modo che le operazioni siano efficienti e la collezione stessa occupi poco spazio di memoria.

Esempio: tipo di dato Dizonario.

Uso di ADT

L'uso di ADT generalizza il concetto di tipo in C, che offre:

- tipi primitivi su cui sono disponibili operazioni predefinite
- tipi definiti dall'utente sui quali non esistono operazioni predefinite.

Un ADT incapsula in un solo involucro una definizione di tipo e le operazioni su di esso, rendendo i tipi definiti dall'utente più omogenei a quelli predefiniti.

L'interfaccia lo caratterizza verso l'esterno.

L'implementazione è una traduzione in termini di linguaggio della dichiarazione della variabile con quelle proprietà e di una funzione per ogni operazione dell'ADT. Una implementazione sceglie una particolare struttura dati per la rappresentazione del tipo, usando i tipi primitivi ed i metodi di strutturazione dati disponibili nel linguaggio scelto.

In generale, per uno stesso ADT sono possibili molteplici realizzazioni alternative basate su **struttura dati** diverse che permettono di implementare le operazioni richieste in modo più o meno efficiente.

Tipo e struttura dati

Il tipo di una variabile è l'insieme dei valori che essa può assumere.

Una struttura dati è la collezione di variabili di vari tipi connesse in vari modi (tipicamente per realizzare il modello matematico di un ADT).

I metodi usati per connettere variabili dei vari tipi e far loro formare una struttura dati sono legati al linguaggio di programmazione (es. array, record, file, struct, …), così come la possibilità di esprimere relazioni fra variabili (puntatori).

Gestione di collezioni di oggetti

Tipo di dato:

- Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca)

Struttura dati:

- Organizzazione dei dati che permette di supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile

Il tipo di dato Dizionario

tipo `Dizionario`:
dati:
un insieme S di coppie $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k).

`delete`$(chiave\ k)$
cancella da S la coppia con chiave k.

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e `null` altrimenti.

Il tipo di dato Pila

tipo `Pila`:
dati:
una sequenza S di n elementi.

operazioni:

`isEmpty`$() \rightarrow result$
restituisce `true` se S è vuota, e `false` altrimenti.

`push`$(elem\ e)$
aggiunge e come ultimo elemento di S.

`pop`$() \rightarrow elem$
toglie da S l'ultimo elemento e lo restituisce.

`top`$() \rightarrow elem$
restituisce l'ultimo elemento di S (senza toglierlo da S).

Il tipo di dato Coda

tipo `Coda`:
dati:
una sequenza S di n elementi.

operazioni:

`isEmpty`$() \rightarrow result$
restituisce `true` se S è vuota, e `false` altrimenti.

`enqueue`$(elem\ e)$
aggiunge e come ultimo elemento di S.

`dequeue`$() \rightarrow elem$
toglie da S il primo elemento e lo restituisce.

`first`$() \rightarrow elem$
restituisce il primo elemento di S (senza toglierlo da S).

Tecniche di rappresentazione dei dati

Rappresentazioni indicizzate:

- I dati sono contenuti in array

Rappresentazioni collegate:

- I dati sono contenuti in record collegati fra loro mediante puntatori

Pro e contro

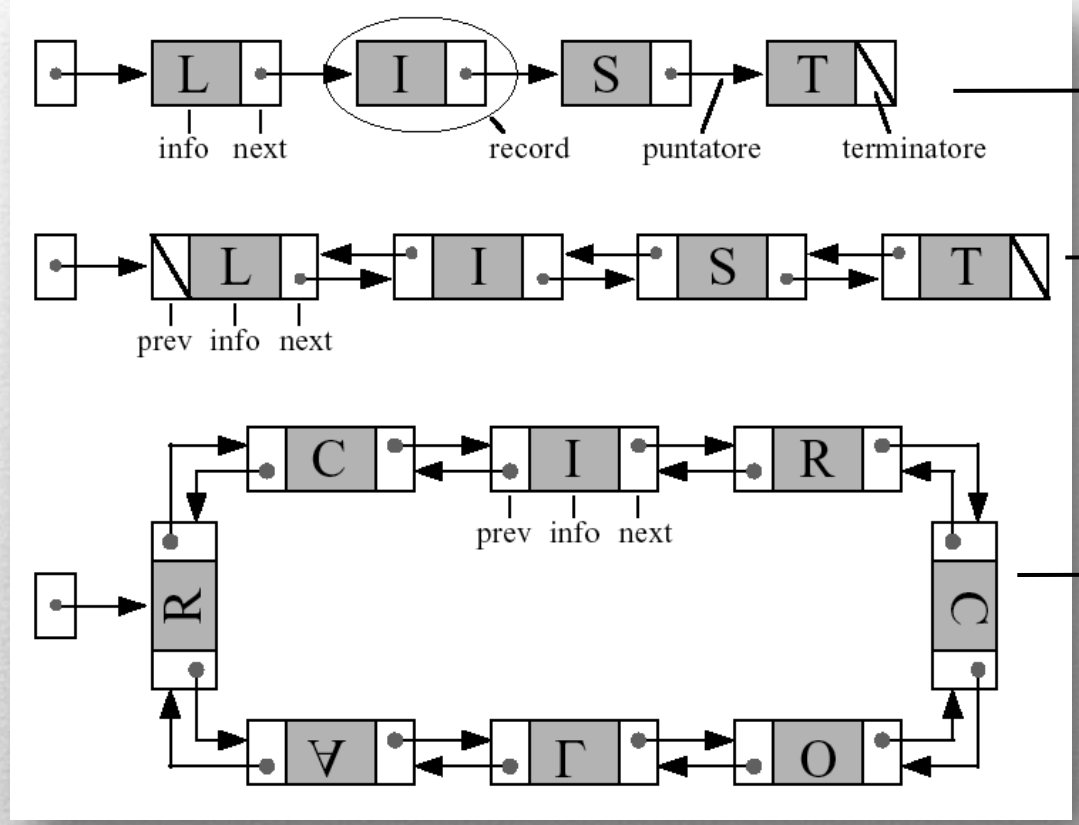
Rappresentazioni indicizzate:

- Pro: accesso diretto ai dati mediante indici
- Contro: dimensione fissa (riallocazione array richiede tempo lineare)

Rappresentazioni collegate:

- Pro: dimensione variabile (aggiunta e rimozione record in tempo costante)
- Contro: accesso sequenziale ai dati

Esempi di strutture collegate



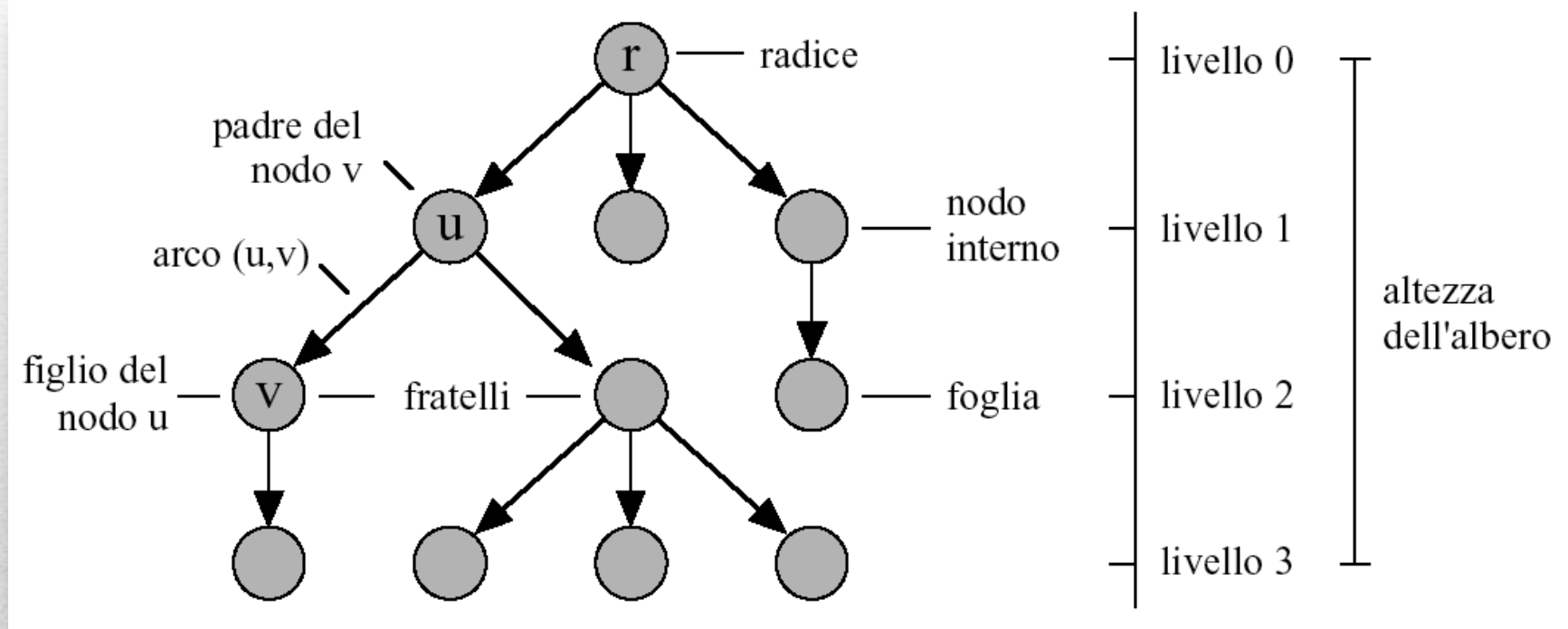
Lista semplice

Lista doppiamente collegata

Lista circolare doppiamente collegata

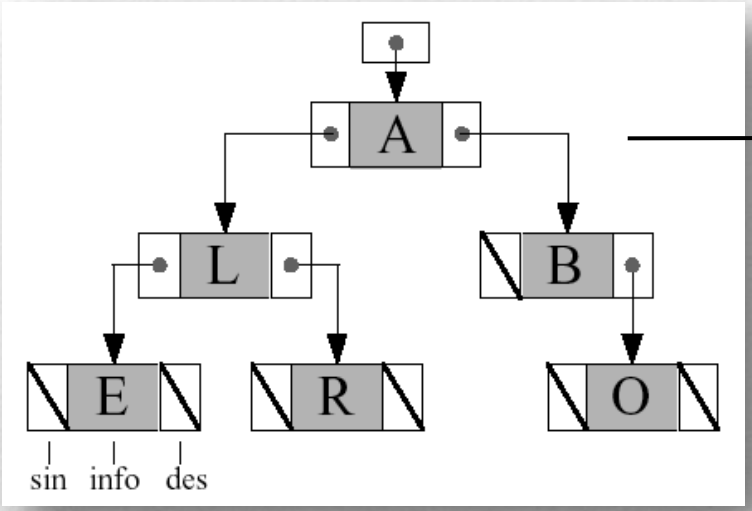
Alberi

Organizzazione gerarchica dei dati:



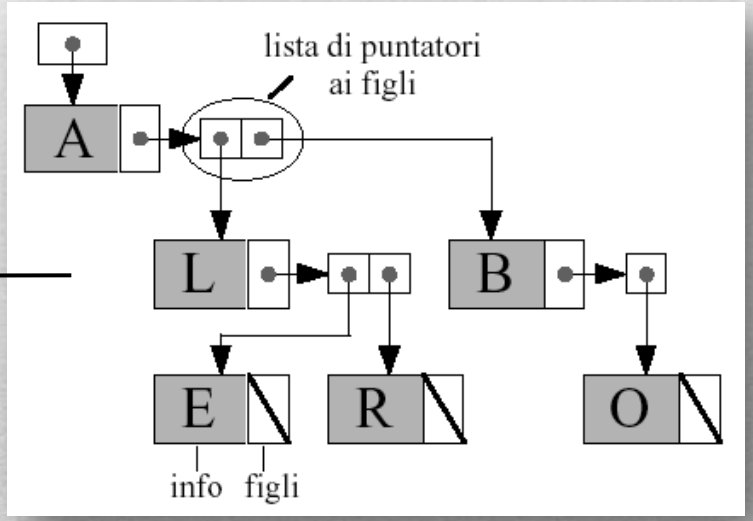
Dati contenuti nei nodi, relazioni gerarchiche definite dagli archi che li collegano.

Rappresentazioni collegate di alberi

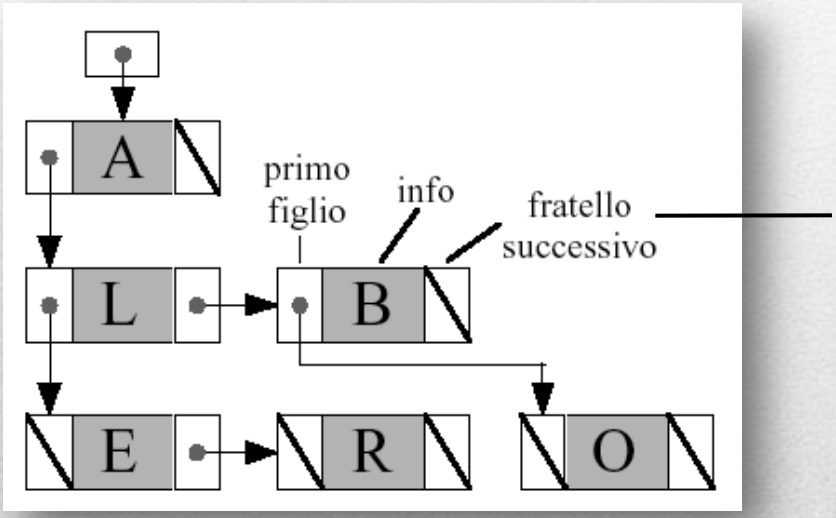


Rappresentazione con puntatori ai figli (nodi con numero limitato di figli)

Rappresentazione con liste di puntatori ai figli (nodi con numero arbitrario di figli)



Rappresentazioni collegate di alberi



Rappresentazione
di tipo primo figlio-fratello successivo (nodi con numero arbitrario di figli)