Algoritmi (modulo di laboratorio)

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Martedì 8.30 - 10.30 in aula 8 Mercoledì 10.30 - 13.30 in aula 309
	Giovedì 16.30 - 18.30 in aula 307 Venerdì 10.30 - 12.30 in aula 4
Ricevimento:	su appuntamento (Dipartimento di Informatica)
E-mail:	roberto.cordone@unimi.it
Pagina web:	http://homes.di.unimi.it/~cordone/courses/2024-algo/2024-algo.html
Sito Ariel:	https://mgoldwurmasd.ariel.ctu.unimi.it

Lezione 1: Problemi, algoritmi, programmi, linguaggio C Milano, A.A. 2023/24

Nozione di algoritmo

Algoritmo è uno strumento formale per risolvere un problema,
cioè un meccanismo che non richiede la comprensione del problema

In questo corso si vuole sviluppare la capacità di

- progettare algoritmi: passare dalla descrizione di un problema alla descrizione di un algoritmo che lo risolve
- analizzare algoritmi: passare dalla descrizione di un problema e di un algoritmo alla dimostrazione
  1. che l'algoritmo risolve il problema (correttezza)
  2. di quanto costa risolvere il problema con l'algoritmo (complessità)

Per farlo, occorre impostare con un po' di rigore le definizioni di

- problema
- risolvere
- algoritmo
- costo

Faremo solo qualche accenno, non una formalizzazione completa

Una definizione informale

Problema è una domanda su un sistema descritto quantitativamente

- insiemi di grandezze numeriche, fisiche, logiche...
- relazioni fra grandezze (ordine, appartenenza, funzioni...)
- operazioni (regole di trasformazione del sistema)

Dunque è un insieme di oggetti matematici

Soluzione è la risposta, descritta quantitativamente, alla domanda

Dunque un secondo insieme di oggetti matematici legato al primo

Se si affrontano problemi concreti

- prima il problema concreto va modellato in un problema astratto
- poi la soluzione astratta va interpretata in una strategia concreta

Nel corso non approfondiremo questi aspetti, ma faremo esempi

Esempi di problemi

1. *Il numero* $2^{17} - 1$ *è un numero primo?*
   - La domanda è definita dal numero $2^{17} - 1$
   - La risposta è *Sì* oppure *No*
2. *In quanto tempo si arriva da qui in Piazza Duomo?*
   - La domanda è definita dalla rete stradale (topologia e tempi)
   - La risposta è un numero (ad es., in minuti)
3. *Quali sono i nuclei familiari residenti a Milano?*
   - La domanda è definita dall'insieme dei cittadini e delle loro relazioni di parentela e convivenza
   - La risposta è una collezione di sottoinsiemi di cittadini
4. *In che ordine sono disposte nel dizionario le parole* zebra, alce, renna, gnu, antilope?
   - La domanda è definita da una sequenza di parole
   - La risposta è una permutazione della sequenza data

Istanze

Capita spesso di porre la stessa domanda su sistemi diversi

- 2 è un numero primo?
- 3 è un numero primo?
- 10^{300} è un numero primo?
- $2^{17} - 1$ è un numero primo?

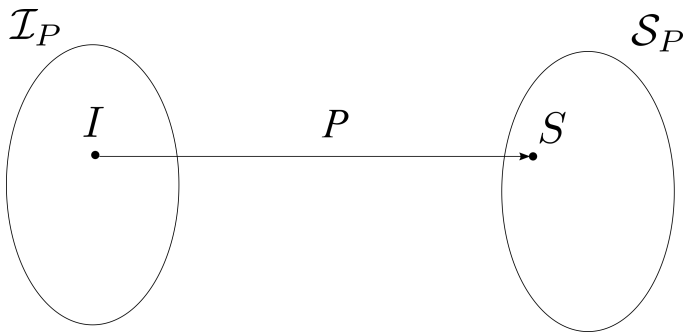
Definiremo

- istanza I la descrizione quantitativa dello specifico sistema su cui si pone una domanda
- soluzione S la descrizione quantitativa della specifica risposta a quella domanda

e riserveremo invece la parola

- problema P alla relazione che lega ogni istanza alla sua soluzione, ovvero un insieme di coppie istanza-soluzione

Problemi e funzioni



- Ogni istanza I rappresenta i dati di uno specifico problema e l'insieme di tutte le istanze del problema è il dominio $\mathcal{I}_P$
- Ogni soluzione S rappresenta i risultati di uno specifico problema e l'insieme di tutte le soluzioni del problema è il codominio $\mathcal{S}_P$
- Il problema P è la corrispondenza fra istanze I e soluzioni S

$$P : \mathcal{I}_P \to \mathcal{S}_P$$

Quindi una funzione che lega insiemi matematici

Esempio: il problema della primalità

Definizione informale: *Il numero naturale n è primo?*

Definizione formale:

$\mathcal{I}_P$ (istanze)	$\mathcal{S}_P$ (soluzioni)
...	...
5	Sì
6	No
7	Sì
8	No
...	...

Introduciamo la classica corrispondenza *Sì* $= 1$ e *No* $= 0$

- le istanze sono tutti i numeri naturali positivi: $\mathcal{I}_P = \mathbb{N}^+$
- le soluzioni sono i numeri naturali 0 e 1: $\mathcal{S}_P = \{0, 1\}$

Il problema della primalità è una funzione $\mathbb{N}^+ \to \{0, 1\}$

Vedremo che tutti i problemi sono funzioni $\mathbb{N} \to \mathbb{N}$*!*

Codifica di un problema

La definizione informale è inadatta a una macchina

Occorre quindi

- codificare l'istanza affinché una macchina possa manipolarla
- codificare la soluzione affinché una macchina possa produrla
- definire il meccanismo di trasformazione da una codifica all'altra

La codifica di un'istanza e della corrispondente soluzione consiste nel

1. sostituire gli oggetti matematici con simboli elementari tratti da un alfabeto finito (di lettere, cifre, segni, ...)
2. raccogliere i simboli in una sequenza finita (stringa)

In pratica, un'istanza viene convertita in un file di testo,
il meccanismo risolutivo lo manipola opportunamente
e produce un file di testo con la soluzione

Quale alfabeto?

Gli alfabeti possibili sono infiniti:

- gli esseri umani (occidentali) usano spesso l'alfabeto ASCII (o l'extended ASCII), che è composto da 128 (o 256) simboli
- i computer usano l'alfabeto binario $\mathcal{A} = \{0, 1\}$

Ogni alfabeto $\mathcal{A}_1$ è traducibile meccanicamente in ogni altro alfabeto $\mathcal{A}_2$

- se $|\mathcal{A}_1| \geq |\mathcal{A}_2|$, si fissa una corrispondenza biunivoca convenzionale fra ogni simbolo di $\mathcal{A}_1$ e una stringa di $\left\lceil \log_{|\mathcal{A}_2|} |\mathcal{A}_1| \right\rceil$ simboli di $\mathcal{A}_2$

In particolare, l'alfabeto ASCII si può convertire in binario per

- trasmettere problemi ai computer
- ricevere soluzioni dai computer

Problemi e funzioni su numeri naturali

Consideriamo un problema non numerico: ordinare un gruppo di lettere

Istanze		Soluzioni	
codifica ASCII	codifica binaria	codifica binaria	codifica ASCII
ab	1100001 1100010	1100001 1100010	ab
eb	1100101 1100010	1100010 1100101	be
vm	1110110 1101101	1101101 1110110	mv
df	1100100 1100110	1100100 1100110	df
...	...	...	...

Una volta codificate in binario

- le istanze sono isomorfe a un sottoinsieme di $\mathbb{N}$ $(12\,514, 13\,026, \ldots)$
- le soluzioni sono isomorfe a un sottoinsieme di $\mathbb{N}$ $(12\,514, 12\,645, \ldots)$

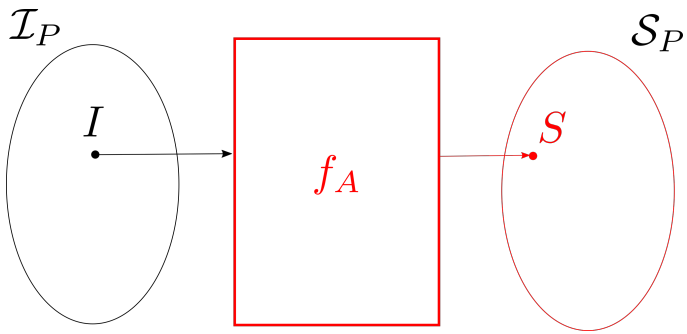
I problemi sono isomorfi a funzioni da numeri naturali a numeri naturali

In generale non sappiamo come calcolarle!

Algoritmi

Algoritmo è una sequenza finita di operazioni elementari meccaniche che trasforma stringhe di simboli in stringhe di simboli tratti da un alfabeto

Quindi un algoritmo è una funzione fra insiemi di stringhe/numeri



Per definire un algoritmo bisogna prima aver definito

- l'alfabeto di lavoro con cui si compongono le stringhe
- le operazioni elementari eseguibili, cioè un modello computazionale (lettura e scrittura di simboli, operazioni su numeri, valutazione di condizioni logiche, ecc. . .)

Nel modulo di teoria sarà presentata la macchina RAM

Algoritmi e problemi

Un algoritmo risolve un problema quando trasforma la stringa che codifica ogni istanza nella stringa che codifica la soluzione corrispondente

$$S_A(I) = S(I) \text{ per ogni } I \in \mathcal{I}_P$$

Un algoritmo computa la funzione che lega istanza e soluzione

Algoritmi e problemi sono legati, ma non identici

- un problema è una funzione
- un algoritmo è una funzione più il modo di computarla

In particolare, esistono

- problemi privi di algoritmi risolutivi
- problemi con algoritmi risolutivi diversi che hanno costo diverso

$$\begin{aligned} A: &\quad 7 + 4n + 2n^2 + n^3 \quad (3 \text{ somme e } 5 \text{ prodotti}) \\ A': &\quad 7 + n[4 + n(2 + n)] \quad (3 \text{ somme e } 2 \text{ prodotti}) \end{aligned}$$

$$S_A(\cdot) \equiv S_{A'}(\cdot) \text{ ma } A \neq A'$$

Linguaggio macchina

Come si realizza un algoritmo?

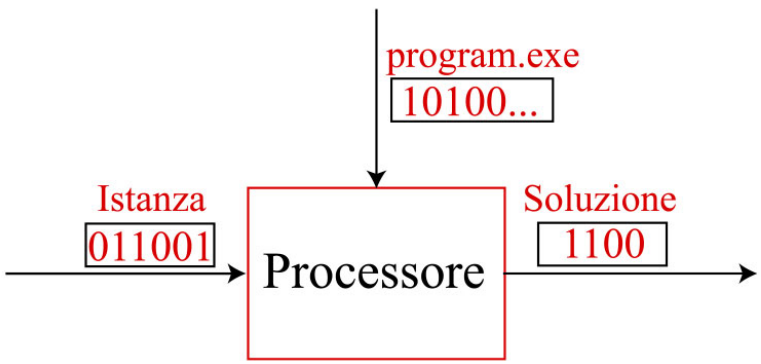
Come si costruisce una macchina che trasforma simboli in simboli?

I processori trasformano stringhe binarie in stringhe binarie,
eseguendo istruzioni espresse come stringhe binarie

Codificando la macchina stessa come stringa binaria si ottiene
un meccanismo di calcolo universale, cioè
si possono realizzare diversi algoritmi sulla stessa macchina

Linguaggio macchina è la codifica binaria delle istruzioni per il processore

Processore


program.exe
10100...
Istanza
011001
Processore
Soluzione
1100

Linguaggio assembly

Il linguaggio macchina ha evidenti svantaggi

- è ostico per un essere umano
- ogni processore ha il suo linguaggio specifico

Linguaggio assembly: per ogni istruzione macchina elementare definisce un'istruzione simbolica corrispondente

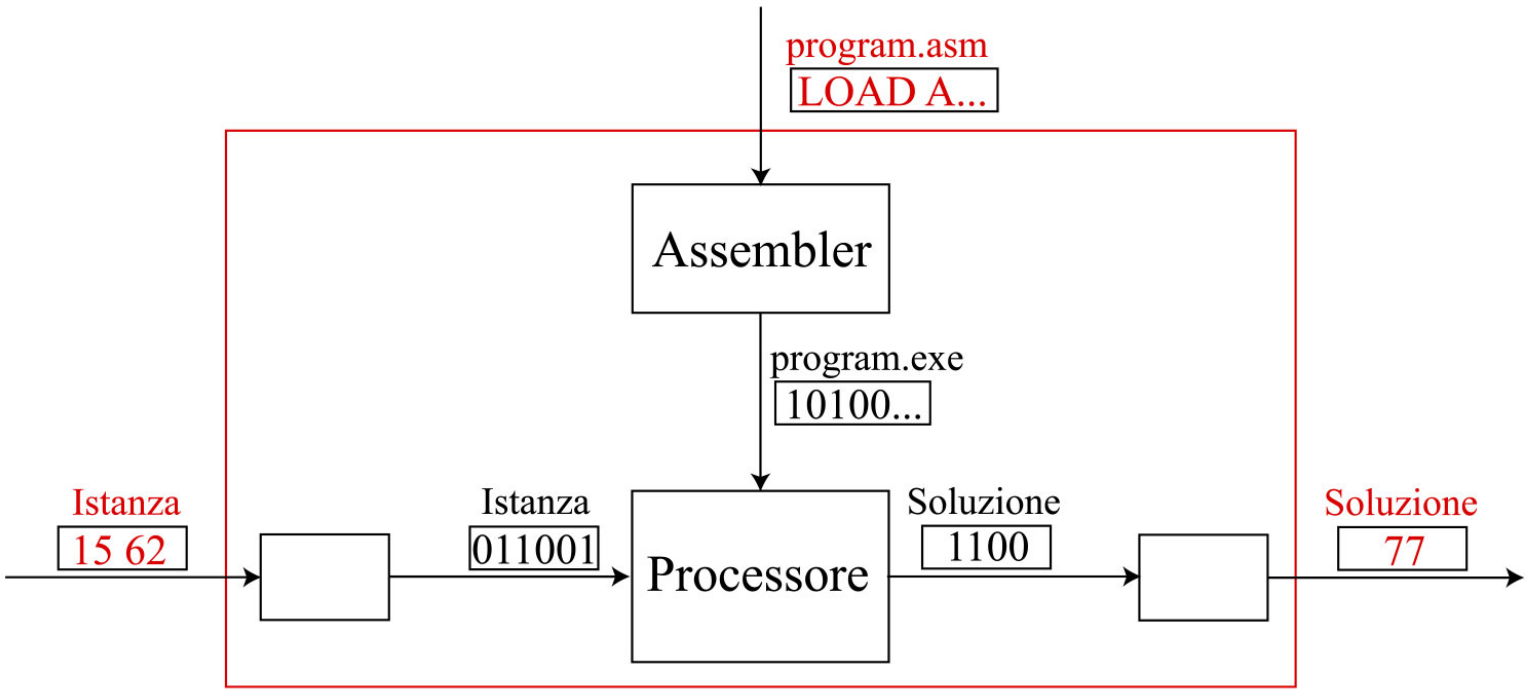
1. si scrive un testo che descrive il programma (codice o listato del programma)
2. un programma (assembler) traduce il codice in linguaggio macchina

LOAD A	1010...00101
LOAD B	1010...01101
ADD	0011...00101
STORE C	1001...10101

Vantaggi:

- il linguaggio assembly è più leggibile per un essere umano
- si può usare un solo linguaggio assembly per diverse macchine purché le macchine abbiano le stesse istruzioni elementari (basta cambiare traduttore)

Assembler


program.asm
LOAD A...
Assembler
program.exe
10100...
Istanza
15 62
Istanza
011001
Processore
Soluzione
1100
Soluzione
77

Linguaggio ad alto livello

Svantaggi

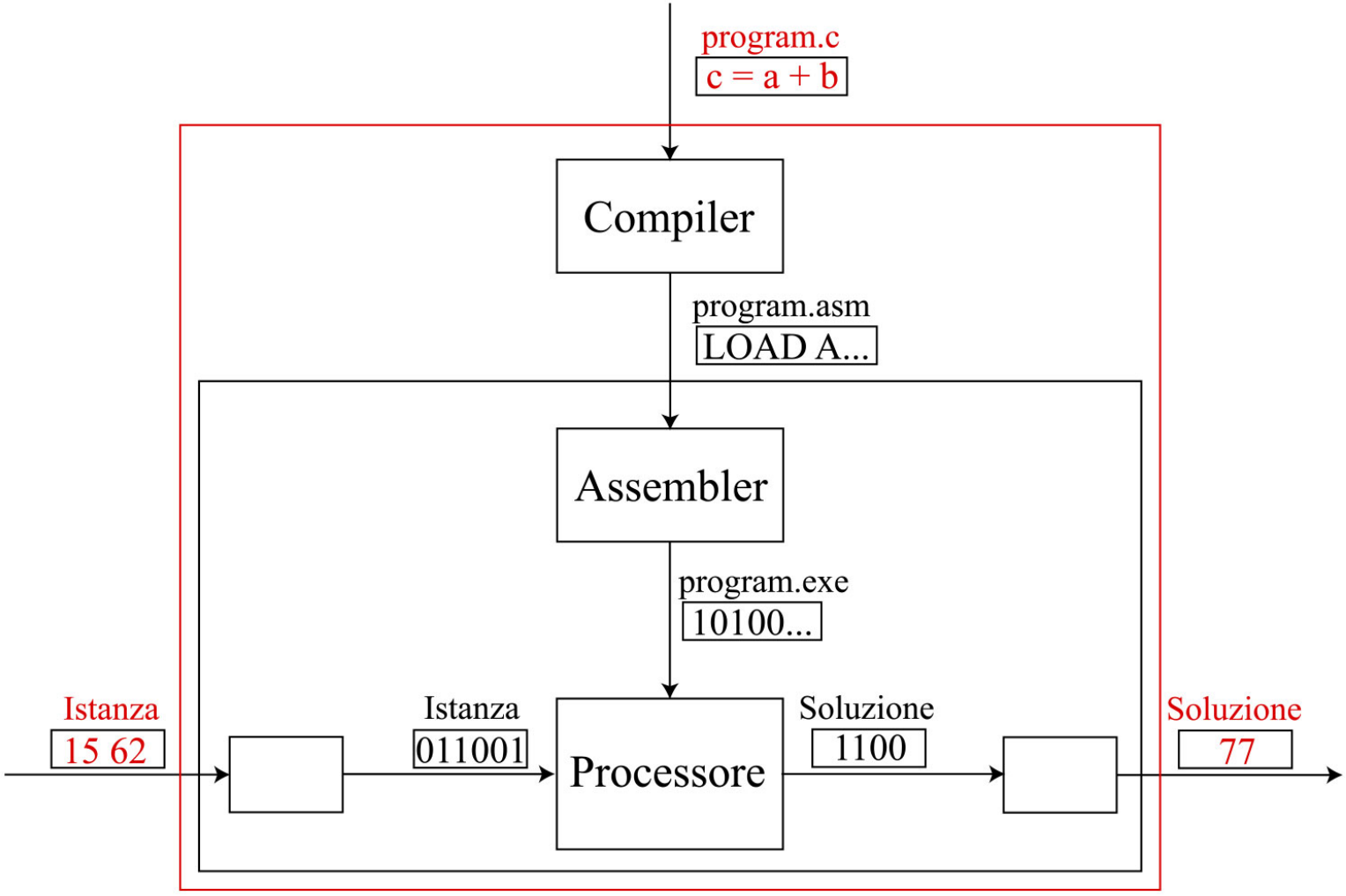
- anche operazioni banali richiedono più istruzioni
- il programma è molto lungo
- il significato del programma non è evidente

Linguaggio ad alto livello: definisce una sola istruzione simbolica per ogni sequenza di istruzioni assembly con uno scopo comune

- si scrive un testo che descrive il programma (codice o listato del programma)
- un programma (compiler) traduce il codice in linguaggio macchina (di solito passando per l'assembly)

Operazione	Alto livello	Assembly
Somma a e b	$c = a + b$	LOAD A LOAD B ADD STORE C

Compilatore



Approccio top-down

L'approccio top-down progetta un algoritmo per un problema

- partendo dai requisiti posti dal problema
- decomponendo il problema in sottoproblemi gerarchicamente (cioè i sottoproblemi in sottosottoproblemi, ecc. . .)
- arrestandosi al livello dei compiti elementari

Compiti elementari sono le operazioni per cui esiste già del codice

Questo approccio si riflette direttamente nella scrittura del codice

Esempio di approccio top-down (1)

Supponiamo di voler far stampare sul video un saluto scelto dall'utente incorniciato con un carattere scelto dall'utente

Questo compito si può scomporre in

1. acquisire dall'utente il carattere desiderato per la cornice (`'*'`)
2. acquisire dall'utente il saluto desiderato (`Hello, world!`)
3. stampare il saluto incorniciato

```
******************
* Hello, world! *
******************
```

Intuiamo che la scomposizione è corretta perché

- ogni sottocompito è chiaro e indipendente dagli altri
- alcuni sottocompiti sono simili, se non identici (acquisizione di uno o più caratteri da tastiera)

Lo stesso avviene ai livelli inferiori

Esempio di approccio top-down (2)

Infatti, le due acquisizioni si scompongono in

a. comunicare all'utente che cosa gli si chiede di fornire
b. acquisire l'informazione richiesta
   (il carattere `cornice` o la riga `saluto`)

La stampa del saluto incorniciato si scompone in

a. determinare la `larghezza` della cornice (lunghezza di `saluto` più 4)
b. stampare la cornice superiore
   (stampare `larghezza` volte il carattere `cornice`)
c. andare a capo
d. stampare la cornice laterale sinistra (stampare il carattere `cornice`)
e. stampare uno spazio separatore (stampare uno spazio)
f. stampare il saluto (stampare la stringa `saluto`)
g. stampare uno spazio separatore (stampare uno spazio)
h. stampare la cornice laterale destra (stampare il carattere `cornice`)
i. andare a capo
j. stampare la cornice inferiore
   (stampare `larghezza` volte il carattere `cornice`)

I compiti ripetuti abbondano!

Esempio di approccio top-down (3)

Quali sono i compiti elementari dove si arresta la scomposizione?
Forse la lettura e la scrittura di caratteri singoli?

Se supponiamo di avere una libreria `advio.h` che fornisca procedure per

- leggere da tastiera singoli caratteri (*per la cornice*)
- leggere da tastiera righe di testo (*per il saluto*)

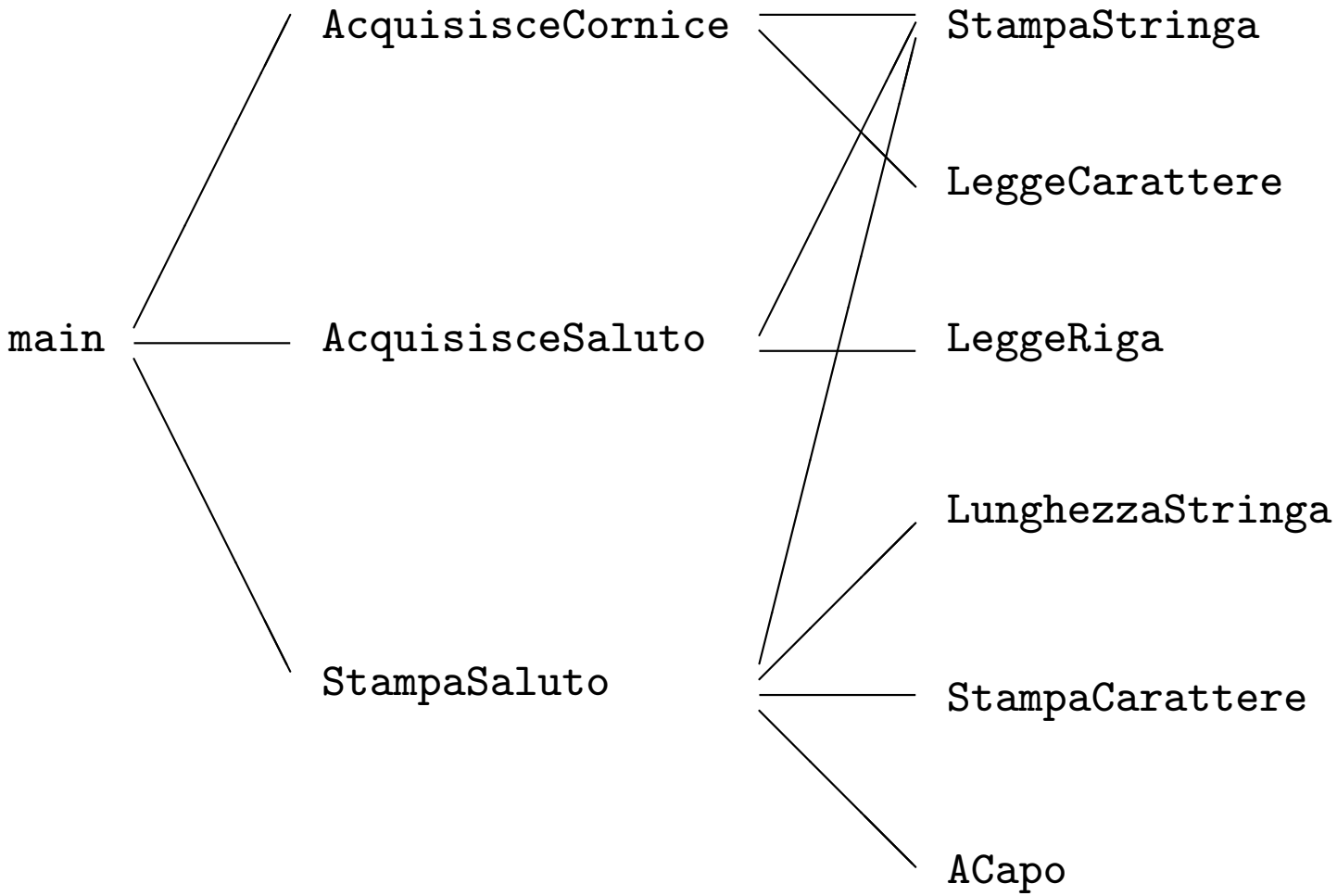
ma anche

- valutare la lunghezza di una stringa di caratteri (*saluto* $\rightarrow$ *cornice*)
- stampare a video caratteri singoli o ripetuti (*cornice o spazio*)
- andare a capo
- stampare a video stringhe di caratteri (*per il saluto*)

possiamo arrestare la scomposizione considerando elementari tali compiti

L'approccio bottom-up consiste nel dotarsi di procedure che consentono di arrestare la scomposizione in anticipo

Esempio di approccio top-down (4)



Struttura modulare

Il progetto top-down si riflette nella struttura modulare del codice

Il codice viene strutturato in modo da essere gestibile

1. si scompone il codice in moduli o blocchi strutturati gerarchicamente (esattamente come il problema è scomposto in sottoproblemi)
2. si rendono i moduli comprensibili adottando convenzioni
   - dichiarazioni che imitano il linguaggio umano
   - nomi autoesplicativi per funzioni, macro e variabili
   - corrispondenza biunivoca fra variabili e oggetti
   - usare spazi, a capi e indentazioni per chiarire il senso
   - commenti (ultima risorsa)
3. si rendono i moduli controllabili
   - definendoli in modo che abbiano poche interazioni fra loro
   - esplicitando i requisiti di ogni modulo (dati e risultati)

Si vuole ridurre la comprensione di un programma a

- comprensione di ciascun modulo
- comprensione dei rapporti fra moduli

Non esistono regole meccaniche: la semplicità è una conquista

Suddivisione del programma

Il codice di un programma complesso viene distribuito in più file perché sarebbe troppo lungo, anche se scritto in linguaggio ad alto livello

- il singolo file è gestibile da un essere umano
- si possono assegnare brani di codice a gruppi indipendenti
- si possono riusare brani di codice in altri programmi senza copiarli (librerie)
- si possono tradurre librerie in linguaggio macchina
  - per evitare di tradurle ogni volta col resto del codice
  - per poterle vendere indipendentemente

Siccome i singoli file non corrispondono a programmi funzionanti

- prima di eseguirli, vanno ricollegati in un codice binario unico
- i singoli file devono contenere indicazioni su come ricollegarli

Compilazione

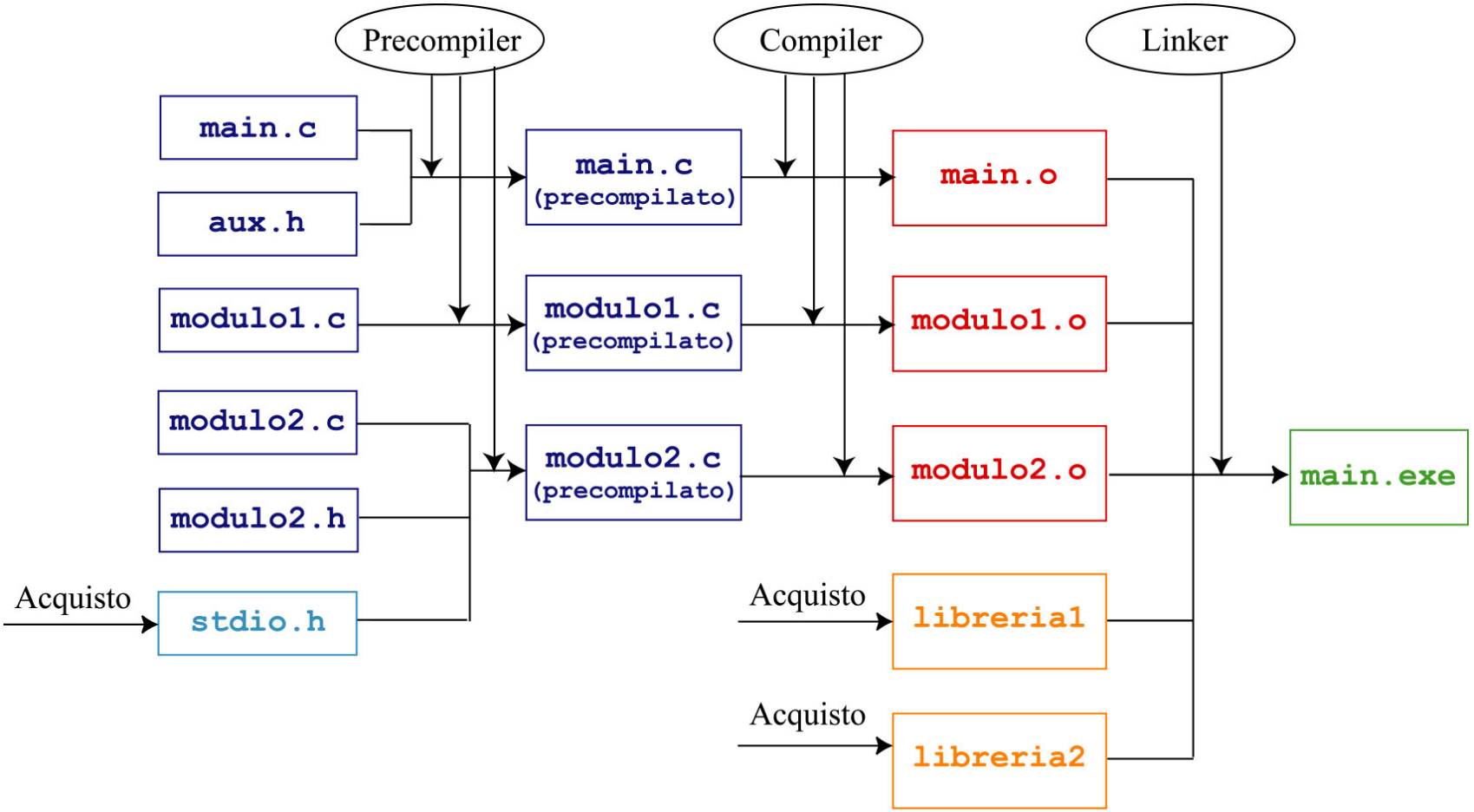
Compilazione è il processo di traduzione

- dal codice in linguaggio ad alto livello (uno o più file di testo)
- al programma in linguaggio macchina (un file eseguibile)

Attraversa tre fasi

1. precompilazione o preprocessing (da codici a codice): modifica o cancella brani di codice ad alto livello e ne fonde diversi in uno solo
2. compilazione (da codice a oggetto): traduce un file codice in un file binario (file oggetto), di solito passando per il codice assembly
3. collegamento o linking (da oggetti a eseguibile): lega diversi file oggetto ed eventuali librerie esterne in un solo programma eseguibile

Compilazione



Struttura dei listati C

Ogni file in linguaggio C (listato) ha una struttura regolare

1. direttive
2. dichiarazioni delle procedure secondarie (prototipi)
3. programma principale (`main`)
   a) parte dichiarativa
   b) parte esecutiva
4. definizioni delle procedure secondarie
   a) parte dichiarativa
   b) parte esecutiva

La struttura è strettamente legata al processo di compilazione

- ciascuna componente può esserci o mancare
  (per es., il `main` compare in un solo file dell'intero progetto)
- l'ordine non è rigoroso, ma violarlo può far fallire la compilazione

Commenti

I commenti sono spiegazioni del codice ad uso degli utenti racchiuse fra `/*` e `*/` (non si possono annidare!)

- sono eliminati dal precompilatore (la macchina non li usa)
- sono essenziali per lavorare in gruppo o a distanza di tempo
- esistono strumenti software che creano automaticamente il manuale del programma partendo dai commenti

Commento è anche il testo compreso fra `//` e la fine della riga
a partire dallo standard C99 (*noi seguiremo quasi in tutto il C89*)

Struttura: direttive (nei file modulo)

Servono a includere file di intestazione e a definire costanti simboliche

```
/* hello.c */


#include <stdlib.h>
#include "advio.h"



char AcquisisceCornice ();
void AcquisisceSaluto (char *saluto);
void StampaSaluto (char *saluto, char cornice);


int main (int argc, char *argv[])
{
  char saluto[ROW_LENGTH];
  char cornice;
```

Struttura: direttive (nei file di intestazione)

Servono anche a non includere più volte lo stesso file di intestazione

```
/* advio.h */


#ifndef __advio_h
#define __advio_h

#define SPAZIO    ' '
#define ROW_LENGTH 256


/* Stampa a video la stringa di caratteri "s" */
void StampaStringa (char *s);

/* Legge da tastiera la stringa di caratteri "s" */
void LeggeRiga (char *s);

...



#endif
```

Direttive

Cominciano con il carattere speciale #

Danno istruzioni al precompilatore

- inclusione di altri file (`#include` *nomefile*):
  si sostituisce la direttiva con l'intero contenuto del file (ricorsivamente)
- espansione di macro (`#define` *macro valore*):
  si sostituisce ogni occorrenza della macro col valore (ricorsivamente)
- compilazione condizionale
  `#ifdef` *macro codice* `#endif`:
  il codice si conserva se la macro è definita, altrimenti si cancella
  `#ifndef` *macro codice* `#endif`:
  il codice si conserva se la macro non è definita, altrimenti si cancella

Per vedere in dettaglio l'azione del precompilatore, si esegue il comando

```
gcc -E hello.c -o hello_precompiled.c
```

Struttura: prototipi delle procedure

```
/* hello.c */


#include <stdlib.h>
#include "advio.h"



char AcquisisceCornice ();
void AcquisisceSaluto (char *saluto);
void StampaSaluto (char *saluto, char cornice);



int main (int argc, char *argv[])
{
  char saluto[ROW_LENGTH];
  char cornice;
```

Struttura: prototipi delle procedure

Presentano le funzioni o procedure: brani di codice abbastanza importanti da avere un nome, dei dati e dei risultati, come i programmi

Il prototipo (dichiarazione) di una funzione ne specifica

1. il tipo del risultato (`void` se non c'è risultato)
2. il nome simbolico usato per chiamarla
3. il tipo e il nome dei dati (parametri di ingresso)

La definizione della funzione (il brano di codice) è in un'altra sezione

Struttura: prototipi inclusi dai file intestazione

I file *header* inclusi dalla direttiva `#include` spesso contengono prototipi di funzioni definite in librerie esterne

```
/* advio.h */

#ifndef __advio_h
#define __advio_h

#define SPAZIO    ' '
#define ROW_LENGTH 256


/* Stampa a video la stringa di caratteri "s" */
void StampaStringa (char *s);


/* Legge da tastiera la stringa di caratteri "s" */
void LeggeRiga (char *s);


...
```

Struttura: main

```
int main (int argc, char *argv[])
{
  char saluto[ROW_LENGTH];
  char cornice;


  /* Acquisisce dall'utente il carattere per la cornice */
  cornice = AcquisisceCornice();

  /* Acquisisce dall'utente il saluto da stampare */
  AcquisisceSaluto(saluto);

  /* Stampa il saluto incorniciato */
  StampaSaluto(saluto,cornice);

  return EXIT_SUCCESS;
}
```

Struttura: main

Uno e uno solo dei moduli C contiene il programma principale (`main`)

La sua intestazione consiste ancora in

1. tipo del risultato (sempre `int`)
2. nome del programma principale
   (sempre `main`, per distinguerlo dalle altre funzioni)
3. tipi e nomi dei dati (sempre `argc` e `argv`)

La definizione è racchiusa fra parentesi graffe ({})

a) parte dichiarativa: introduce le variabili,
   cioè gli oggetti manipolati dal programma

b) parte esecutiva: introduce le istruzioni,
   cioè le operazioni compiute dal programma

Struttura: parte dichiarativa

```
int main (int argc, char *argv[])
{
  char saluto[ROW_LENGTH];
  char cornice;

  /* Acquisisce dall'utente il carattere per la cornice */
  cornice = AcquisisceCornice();

  /* Acquisisce dall'utente il saluto da stampare */
  AcquisisceSaluto(saluto);

  /* Stampa il saluto incorniciato */
  StampaSaluto(saluto,cornice);

  return EXIT_SUCCESS;
}
```

Struttura: parte dichiarativa

Le variabili sono risultati parziali; di ognuna si specifica

- il tipo, che identifica l'insieme dei valori che può assumere, le operazioni che si possono compiere con essa e lo spazio che occupa in memoria
- il nome, cioè un identificatore simbolico usato per manipolarla

Struttura: parte esecutiva

```
int main (int argc, char *argv[])
{
  char saluto[ROW_LENGTH];
  char cornice;


  /* Acquisisce dall'utente il carattere per la cornice */
  cornice = AcquisisceCornice();

  /* Acquisisce dall'utente il saluto da stampare */
  AcquisisceSaluto(saluto);

  /* Stampa il saluto incorniciato */
  StampaSaluto(saluto,cornice);

  return EXIT_SUCCESS;
}
```

Struttura: parte esecutiva

Le istruzioni possono essere

- operazioni definite nel linguaggio (per es., =)
- funzioni tratte da librerie standard (per es., `printf`)
- funzioni definite dall'utente (per es., `StampaStringa`)

Ogni istruzione termina con un punto e virgola (;)

La dichiarazione di una variabile o di una procedura è un'istruzione

- aggiunge il nome della variabile o procedura alla tabella dei simboli utilizzabili dal programma
- quindi, termina con ;

Struttura: definizioni delle procedure

```
/* Stampa il saluto incorniciato */
void StampaSaluto (char *saluto, char cornice)
{
  int larghezza;

  /* Determina la lunghezza della cornice */
  larghezza = LunghezzaStringa(saluto) + 4;

  /* Stampa la cornice superiore */
  StampaCarattere(cornice,larghezza);
  ACapo();

  /* Stampa la cornice laterale sinistra */
  StampaCarattere(cornice,1);
  StampaCarattere(SPAZIO,1);

  /* Stampa il saluto */
  StampaStringa(saluto);

  /* Stampa la cornice laterale destra */
  StampaCarattere(SPAZIO,1);
  StampaCarattere(cornice,1);
  ACapo();

  /* Stampa la cornice inferiore */
  StampaCarattere(cornice,larghezza);
}
```

Struttura: definizioni delle procedure

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* Stampa a video la stringa di caratteri "s" */
void StampaStringa (char *s)
{
  printf("%s",s);
}

...

/* Stampa a video il carattere "c" ripetuto "num" volte */
void StampaCarattere (char c, int num)
{
  int i;

  for (i = 1; i <= num; i++)
    printf("%c",c);
}

...
```

Struttura: definizioni delle procedure

La definizione è il brano di codice preannunciato dal prototipo

Ha la stessa struttura del main

- intestazione: copia del prototipo (senza ;)
- parte dichiarativa: tipo e nome delle variabili
- parte esecutiva: istruzioni

Compilazione

Per trasformare un programma C in eseguibile si usa il programma `gcc`

- `gcc -E modulo.c -o modulo_precompilato.c`
  esegue solo la precompilazione su un modulo C
  Non si usa praticamente mai
- `gcc -c modulo.c -o modulo.o`
  esegue precompilazione e compilazione su un modulo C
  producendo un file oggetto
- `gcc modulo.c modulo2.c modulo3.c -o prog`
  esegue precompilazione, compilazione e collegamento sui moduli
  producendo un file eseguibile (*in Windows*, `prog.exe`)

Errori e avvertimenti

Il compilatore fa un controllo sugli errori sintattici del codice e li segnala

Alcuni costrutti sintattici possono essere corretti sintatticamente,
ma sospetti, perché spesso abbinati a errori semantici
(*per es., usare variabili non inizializzate*)

Il programma gcc segnala alcuni di questi costrutti con degli *warning*:
conviene tenerli d'occhio e risolverli per

- ridurre gli errori semantici
- aumentare la portabilità

Le seguenti opzioni regolano gli avvertimenti

- `-Wall`: indica di stampare tutti i possibili messaggi di avvertimento
- `-pedantic`: indica di richiedere un'aderenza stretta al C standard