Funzioni

Modularizzazione

Quando il progetto diviene complesso allora, per poter essere gestito, è necessario che venga **modularizzato**:

- il progetto viene strutturato in **parti separate**
- si stabiliscono **relazioni precise** tra le parti

Qualità di una modularizzazione

- **livello di dettaglio** dei sottoproblemi deve scaturire da scelte di progetto
- ogni sottoproblema deve essere **ben caratterizzabile** e risolvibile in modo indipendente
- le soluzioni dei sottoproblemi devono essere **combinabili** in modo semplice

Una buona modularizzazione si ottiene utilizzando il concetto di **astrazione**:

- ci si focalizza sugli aspetti essenziali del problema
- si ignorano aspetti non rilevanti rispetto all'obiettivo

Tipi di astrazione

Astrazione sui dati: attraverso uso di **tipi di dato astratti** (li vediamo più avanti)

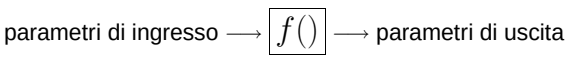
- collezioni di oggetti singoli
- operazioni con le quali operare su questi oggetti

Astrazione funzionale: ci si concentra sul "cosa" e non sul "come"

Noi trattiamo soprattutto la **modularizzazione per astrazione funzionale**

- supportata dai linguaggi imperativi tradizionali (C, Pascal, Fortran)
- realizzata in C attraverso la nozione di **funzione**

Una funzione può essere vista come una **scatola nera**:



- una funzione risolve un sottoproblema specifico
- attraverso i parametri la funzione scambia informazioni con altre funzioni

Le funzioni C

Esempio: Progettare un'interfaccia utente per la stampa di figure geometriche, in cui l'utente può scegliere:

1. la forma della figura $\Longrightarrow$ una funzione per ogni figura
2. la dimensione
3. il carattere di riempimento
4. di quanto spostare a destra la figura

Il programma può essere realizzato a diversi **livelli di generalità**, legati ad un'**astrazione crescente** del concetto di figura.

A questo corrisponde un livello crescente di **parametrizzazione** delle funzioni di stampa delle figure:

- livello (1) non è parametrico
- livello (2) è parametrico rispetto alla dimensione
- livello (3) è parametrico anche rispetto al carattere
- livello (4) è parametrico anche rispetto allo spostamento

Consideriamo prima solo il livello (1)

algoritmo stampa di figure a livello (1)
  do stampa un messaggio
    leggi un carattere
    switch carattere letto
      case 't' : stampa un triangolo
      case 'q' : stampa un quadrato
      case 'f' : stampa un saluto
  while il carattere letto è diverso da 'f'

Caratteri in C: si utilizza il tipo primitivo `char`

- ogni carattere è rappresentato dal suo codice
- i caratteri possone essere **usati come gli interi** (un carattere coincide con il codice che lo rappresenta)
- nei programmi C, un carattere viene racchiuso tra una coppia di apici singoli
  Es.: `'A'`, `'X'`, `'b'`, `'3'`, `'0'`, `';'`, `' '`
- per l'input/output di un carattere si usa lo specificatore di formato "%c"

Implementazione: stampa messaggio invece della figura: file `funzioni/figure0.c`

*Sintassi della **definizione di funzione***

intestazione blocco

dove

- *blocco* costituisce il **corpo della funzione**
- *intestazione* costituisce l'**intestazione della funzione** ed ha la seguente forma:

  identificatore-tipo identificatore (lista-parametri-formali)

  – *identificatore-tipo* specifica il **tipo del valore di ritorno**, ovvero il tipo del risultato restituito alla funzione chiamante (se manca viene assunto `int`)
  – *identificatore* specifica il **nome** della funzione ed è un qualsiasi identificatore C valido
  – *lista-parametri-formali* serve a passare informazioni dalla funzione chiamante a quella chiamata e viceversa:
    * è una lista di dichiarazioni di parametri (tipo e nome) separate da virgola
    * ogni parametro è una **variabile**
    * la lista di parametri può essere vuota

Esempi di intestazioni di funzione

```
char LeggiCarattereNonSpazio() { ... }
int MassimoComunDivisore(int a, int b) { ... }
double Potenza(double x, double y) { ... }
```

N.B. **Non** ci deve essere un "`;`" tra l'intestazione ed il corpo.

Se si omette il tipo di un parametro viene assunto per default il tipo `int`.

Attenzione: `double Potenza(double a, b) { ... }`
equivale a `double Potenza(double a, int b) { ... }`
e non a `double Potenza(double a, double b) { ... }`

N.B. **Non** si possono definire funzioni all'interno di altre funzioni. Quindi tutte le funzioni sono definite allo stesso livello.

*Sintassi della **attivazione di funzione** (detta anche invocazione o chiamata)*

```
identificatore (lista-parametri-attuali)
```

- *identificatore* specifica il nome della funzione
- *lista-parametri-attuali* è una lista di **espressioni** separate da virgola
- i parametri attuali devono corrispondere in numero e tipo ai parametri formali

Semantica di una attivazione di funzione B da una funzione A

- una attivazione di funzione è un'espressione
- viene sospesa l'esecuzione di A e si passa ad eseguire le istruzioni di B (a partire dalla prima)
- quando termina l'esecuzione di B, prosegue l'esecuzione di A dal punto in cui B era stata attivata

N.B. La definizione di una funzione non comporta la sua attivazione.

Prima di poter essere usata (ovvero attivata) una funzione deve essere stata **definita** (o **dichiarata** — vediamo più avanti cosa vuol dire dichiarare una funzione).

Variabili locali

Il **corpo della funzione** è un blocco $\Longrightarrow$ può contenere dichiarazioni di variabili:

- sono **locali** alla funzione (nozione a compile-time)
- hanno **tempo di vita** limitato alla durata dell'attivazione (nozione a run-time)

Regole di visibilità degli identificatori

- un identificatore dichiarato nel corpo di una funzione è detto **locale** alla funzione e **non è visibile all'esterno** della funzione ma solo nel corpo
- in realtà vale una regola più generale: un identificatore dichiarato in un blocco B è visibile
  - nel blocco B (ovvero fino a "}")
  - e in tutti i blocchi interni a B, a meno che non venga ridichiarato.
- un indentificatore dichiarato fuori da qualsiasi blocco è visibile nel file

N.B. La **visibilità** di un identificatore è un concetto rilevante a **compile time**.

Esempio: file `funzioni/scope.c`

Tempo di vita (o esistenza) di una variabile

Le variabili locali (ovvero le locazioni di memoria associate) vengono

- create al momento dell'attivazione di una funzione
- distruttre al momento dell'uscita dall'attivazione

Segue che:

- la funzione chiamante non può fare riferimento ad una variabile locale alla funzione chiamata
- ad attivazioni successive corrispondono variabili (locazioni di memoria) diverse

N.B. Il **tempo di vita** di una variabile è un concetto rilevante a **run time**.

Esempio: Stampa figure a livello (1): file `funzioni/figure1.c`

Esercizio: Scrivere un programma che stampa un rettangolo di asterischi a larghezza fissa e altezza variabile, utilizzando una funzione per la stampa di una riga di asterischi a lunghezza fissa.

Soluzione: file `funzioni/rettang1.c` (senza variabili locali) e `funzioni/rettang2.c` (con variabili locali)

Parametri

Esempio: Progettare un'interfaccia per la stampa di figure a livello (2) (utente può specificare la dimensione).

algoritmo stampa di figure a livello (2), (3) e (4)
  do stampa un messaggio
    leggi un carattere
    if il carattere letto è $\neq$ '`f`'
    then acquisisci ulteriori informazioni
      (ad es. per (3), leggi la dimensione e il carattere di riempimento)
      switch carattere letto
        case '`t`' : stampa un triangolo usando le ulteriori informazioni
        case 'q' : stampa un quadrato usando le ulteriori informazioni
      else stampa un saluto
  while il carattere letto è diverso da '`f`'

Per passare le ulteriori informazioni da `main` alle funzioni di stampa è necessario utilizzare dei **parametri**:

- permettono uno scambio di dati da chiamante a chiamato (e viceversa)
- nell'intestazione: lista di **parametri formali** (con tipo associato) — sono simili a variabili locali, ma vengono inizializzati
- nell'attivazione: lista di **parametri attuali** — possono essere delle espressioni

Al momento dell'attivazione ogni **parametro formale viene inizializzato al valore del corrispondente parametro attuale**.

$\Longrightarrow$ Il valore del parametro attuale viene **copiato** nella locazione di memoria del corrispondente parametro formale.

Esempio: Interfaccia per la stampa di figure a livello (2): file `funzioni/figure2.c`

Esercizio: Scrivere un programma per la stampa di figure geometriche a livello (4).

Soluzione: file `funzioni/figure4.c`

Esercizio: Scrivere un programma per la stampa di un rettangolo di altezza, larghezza e carattere di riempimento variabile. Il programma deve utilizzare una funzione di stampa di una riga del rettangolo.

Soluzione: file `funzioni/rettang3.c`

Esercizio: Scrivere un programma per la stampa di figure geometriche a livello (4), utilizzando una funzione per la stampa di una sequenza di caratteri (con lunghezza e carattere da stampare come parametri).

Soluzione: file `funzioni/figure5.c`

Funzioni che restituiscono un valore

Una funzione che restituisce un valore ha tipo di ritorno diverso da `void`.

Esempio: Funzione che restituisce il massimo tra due interi.

```
int max(int m, int n)
{
  if (m >= n)
    return m;
  else
    return n;
}
```

Attivazione di `max`, ad esempio da `main`:

```
int main(void)
{
  int i, j, massimo;
  scanf("%d%d", &i, &j);
  massimo = max(i,j);
  printf("massimo = %d\n");
  return 0;
}
```


Nel corpo **deve** esserci l'istruzione `return` *espressione*`;`

- restituisce il valore calcolato dalla funzione, che deve essere del tipo del valore di ritorno della funzione
- ritorna il controllo alla funzione chiamante

Esempio:

```
int max(int m, int n)
{
  if (m >= n)
    return m;
  else
    return n;
  printf("pippo");   /* non viene mai eseguita */
}
```

L'istruzione `return` può essere usata anche per funzioni void.

```
void f(int i)
{ ...
  if (i >= 0) return;
  printf("valore negativo");   /* non viene eseguita se i>=0 */
}
```


Esempio: Conversione da numero romano a intero.

- ingresso: sequenza di "cifre romane" terminata da '`\n`'
- uscita: intero corrispondente

Facciamo uso di una funzione `Romano2Intero` che converte una singola cifra romana.

Variante 1: assumiamo che le cifre romane compaiano solo in ordine decrescente
Es.: `MMCLXVII` va bene, mentre `MCMX` no (perché `CM` e' decrescente)

Implementazione: file `funzioni/romani2.c`

Variante 2: sequenza di cifre romane qualsiasi (purché corretta)
Quando leggiamo una cifra

- dobbiamo aggiungerla alla somma corrente se è $\geq$ della cifra successiva
  Es.: `MMC`
- dobbiamo sottrarla dalla somma corrente se è $<$ della cifra successiva
  Es. `MCM`

$\Longrightarrow$ prima di decidere dobbiamo leggere la cifra successiva

Nell'algoritmo usiamo una *somma* corrente e due variabili *cifra_corrente* e *cifra_successiva* che mantengono le ultime due cifre romane lette.

algoritmo conversione da numero romano a intero
  inizializza *somma* a 0
  leggi *cifra_corrente*
  if *cifra_corrente* $\neq$ '\n'
  then leggi *cifra_successiva*
    while *cifra_successiva* $\neq$ '\n'
    do if valore di *cifra_corrente* $\geq$ valore di *cifra_successiva*
      then aggiungi valore di *cifra_corrente* a *somma*
      else sottrai valore di *cifra_corrente* da *somma*
      poni *cifra_corrente* pari a *cifra_successiva*
      leggi *cifra_successiva*
    aggiungi valore di *cifra_corrente* a *somma*
  stampa *somma*

Implementazione: per **esercizio**: file `funzioni/romani3.c`

Dichiarazioni di funzione (o prototipi)

I parametri attuali nell'attivazione di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.

Dobbiamo permettere al compilatore di fare questo controllo
$\Longrightarrow$ prima dell'attivazione deve conoscere l'intestazione della funzione.

Due possibilità:
1. la funzione è stata **definita** prima
2. la funzione è stata **dichiarata** prima

Sintassi di una **dichiarazione di funzione** (o **prototipo**): `intestazione;`

ovvero: `tipo-di-ritorno nome-funzione (lista-parametri-formali);`

- c'e' un "`;`" finale al posto del blocco
- nella lista di parametri formali può anche mancare il nome dei parametri — interessa solo il tipo
- il compilatore usa la dichiarazione per controllare che l'attivazione sia corretta
- dopo deve esserci una definizione della funzione coerente con la dichiarazione

Ordine di dichiarazioni e funzioni

Ogni funzione deve essere stata dichiarata o definita prima di essere usata.

È pratica comune specificare in quest'ordine:
1. dichiarazioni di tutte le funzioni (tranne `main`)
2. definizione di `main`
3. definizioni delle rimanenti funzioni

In questo modo ogni funzione è stata dichiarata prima di essere usata e l'**ordine** in cui mettiamo dichiarazioni e definizioni è **irrilevante**.

Esempio:

```
int Romano2Intero(char);

int main(void) { ... }

int Romano2Intero(char ch) { ... }
```

File header (o di intestazione)

Ogni libreria standard ha un corrispondente file header, che contiene

- definizioni di costanti
- definizioni di tipo
- dichiarazioni di tutte le funzioni della libreria

Esempi:

`<stdio.h>`	input/output
`<stdlib.h>`	allocazione della memoria, numeri casuali, utilità generali
`<string.h>`	manipolazione di stringhe
`<limits.h>`	limiti del sistema per valori interi
`<float.h>`	limiti del sistema per valori reali
`<math.h>`	funzioni matematiche

...

Possono essere scritti anche per funzioni sviluppate da noi (hanno estensione .h).

Funzioni della libreria matematica

Per poterle utilizzare bisogna specificare `#include <math.h>`

Sia argomenti che valore di ritorno sono reali in doppia precisione, ovvero di tipo `double` (e non `float`).

Funzioni disponibili:

`sqrt(x)`	radice quadrata
`exp(x)`	e^x
`log(x)`	logaritmo naturale
`log10(x)`	logaritmo in base 10
`fabs(x)`	valore assoluto
`ceil(x)`	arrotonda all'intero più piccolo $\geq x$
`floor(x)`	arrotonda all'intero più grande $\leq x$
`pow(x,y)`	x^y
`fmod(x,y)`	resto di x/y (in virgola mobile)
`sin(x)`, `cos(x)`, `tan(x)`	trigonometriche (x espresso in radianti)

Esercizio: Calcolo del numero delle combinazioni di n oggetti presi r ad r, ovvero

$$\frac{n!}{r! \cdot (n-r)!}$$

utilizzando una funzione per il calcolo del fattoriale.

Soluzione: file `funzioni/combi.c`

Esercizio: Modularizzare i programmi visti finora a lezione o dati come esercizio attraverso l'introduzione di opportune funzioni.