funzioni

Astrazione

- Consiste nell'ignorare i dettagli e concentrarsi sull'essenziale: in particolare ci consente di utilizzare oggetti complicati con uno sforzo limitato (lettori di CD, automobili, computer)
- Nel nostro caso si tratta di utilizzare codice esistente, sapendo cosa faccia, come invocarlo, ma senza alcun bisogno di sapere come lo faccia

Uso di funzioni

- Per molti scopi possiamo utilizzare una funzione esistente, abbiamo solo bisogno di conoscere
  - Il suo *prototipo*
  - Le *precondizioni*
  - Le *postcondizioni*
- Mentre possiamo ignorare completamente
  - la sua *implementazione*
- La forma generale di una chiamata ad una funzione è un'espressione del tipo

  nome_funzione (lista_di_argomenti)

Precondizioni e Postcondizioni

- Precondizione: ciò che la funzione richiede
- Postcondizione: ciò che la funzione farà se è soddisfatta la precondizione

Prototipo di funzione

- dichiarazione completa di una funzione senza la sua implementazione. I file header (nome.h) contengono di solito una lista di prototipi
- sintassi:

```
tipo nome_funzione (lista_parametri);
```

Dove `lista_parametri` consiste in zero o più parametri separati da virgole

Un parametro può essere

- tipo
- tipo identificatore
- tipo & identificatore
- const altro_parametro (ne riparleremo)

Esempio: elevazione a potenza

- Per elevare un numero x alla potenza y ci basta sapere che nelle librerie del C (che carichiamo specificando -lm nella linea del comando gcc) c'è una funzione che fa questo lavoro: `pow`.
- Per usarla dobbiamo sapere quali informazioni passarle (precondizioni), che cosa ci restituirà (postcondizioni), e di che tipo siano gli oggetti scambiati (informazione fornita dal prototipo)
- `pow` acetta due argomenti di tipo `double`, x e y, e restituisce un `double`, "*x alla y*". Il suo prototipo è

```
double pow(double x, double y);
```

Esempi

- Prototipi:
  - double fabs(double);
  - double sqrt(double x);
  - double pow(double x,double y);
- Uso:
  - a=fabs(-1);
  - b=sqrt(4.0);
  - c=pow(0.5,2);

Informazioni fornite dal prototipo:

- il tipo dell'informazione restituita dalla funzione (void se la funzione non ritorna nulla)
- il nome della funzione
- il numero di argomenti da usare nella chiamata
- il tipo degli argomenti

Moduli

- Un modulo è una racolta di cose collegate tra di loro, quali funzioni, costanti e strutture dati.
- Ad esempio il modulo math rappresenta una racolta di funzioni matematiche e di costanti utili, come M_PI che vale π.
- Per usare le funzioni definite in un modulo bisogna
  - Includere il file header del modulo per avere i prototipi delle funzioni
  - Caricare la libreria del modulo durante il link

Espressioni con funzioni

- Le funzioni hanno priorità e vengono valutate prima degli operatori aritmetici e logici.
- Qualora necessario gli argomenti di una funzione vengono calcolati prima di invocare la funzione.
- Esempio: quanto vale

  50.0 / pow((1.5 + 3.5), 2)?

Errori

- Le chiamate a funzioni possono introdurre errori
  - alcuni dei quali vengono trovati già dal compilatore
  - altri si manifestano solo durante l'esecuzione.
- Esempi:
  1. **numero sbagliato di argomenti: sin( ), sqrt(2,3), pow(2.0)**
  2. **tipo di argomento sbagliato: sqrt("pippo")**
  3. **funzione non definita per l'argomento usato: sqrt(-1.0)**
  4. **risultato non definito (overflow o underflow): pow (99999999., 99999999.)**

Funzioni private

- Scrivere il vostro codice facendo un uso abbondante di funzioni lo rende
  - Di più facile lettura
  - Facilmente modificabile
- Inoltre le vostre funzioni, eventualmente raccolte in un modulo, possono essere riutilizzate in altri programmi

Scrittura di funzioni

- L'implementazione delle nuove funzioni può essere scritta all'interno dello stesso file che contiene il programma (main) ma per la modularità del codice è preferibile dedicarle uno o più altri file.
- Assumiamo di avere il main in prog.c. Se questo fa uso di funzioni che abbiamo implementato nel file funz.c è necessario fornire i prototipi delle funzioni in un file funz.h, che dovrà essere incluso all'inizio di prog.c.

esempio: factorial

una funzione privata per il calcolo del fattoriale

- argomenti: un intero
- nome: factorial
- tipo del risultato ritornato: unsigned long long int
- nell'*header* file `fact.h` dovremo dichiarare la funzione mediante il prototipo

```
unsigned long long int factorial(int n);
```

(n e' un nome simbolico che nel protipo puo' esssere omesso)

- nel file `fact.c` inseriremo il corpo della funzione

```
unsigned long long int factorial(int n){
....
}
```

in questo caso n va specificato in quanto verra' usato nel corpo della funzione e non sara' necessario ridichiararlo

fact.h

```
#ifndef FACT_H
#define FACT_H
 unsigned long long int factorial(int n);
#endif
```

fact.c

```
#include  "fact.h"
/* notare i doppi apici invece di < > */

unsigned long long int factorial(int n ) {
  int i;
  unsigned long long int p = 1;
  for (i = 1; i <= n; i++) {
    p *= i;
  }
  return p;
/* questa istruzione restituisce il
controllo alla funzione chiamante
passandole il risultato che deve essere
dello stesso tipo di questa funzione */
}
```

chiamata a factorial

```
#include  "fact.h"
main() {
 int num;
 printf("inserisci num<21\n");
 scanf("%d",&num);
 if(num<21) {   /* precondizione!!! */
  printf("%d!=%llu\n",num,factorial(num));
  /* NB il valore di num sara' assegnato alla
  variabile n all'interno di factorial */
 } else {
  printf("valore di num %d non valido\n",num);
 }
}
```

la funzione `main`

- si puo' definire senza argomenti, main( ) o main(void), o con due argomenti che consentono di passare variabili dall'esterno del programma
- in questo secondo caso l'uso degli argomenti dipende dal sistema operativo, nei sistemi UNIX

```
int main(int argc, char * argv[]){
...
}
```

dove argc e' un intero corrispondente al numero di stringhe di caratteri trasmesse al programma + 1 e argv[] e' un vettore di puntatori a stringhe di caratteri, argv[0] e' il nome dell'eseguibile, le successive contengono argomenti o opzioni per il programma:

```
myprogram.exe random.dat -p=1
```

ricorsione

- il C consente la ricorsione ovvero la chiamata di una funzione all'interno del corpo della funzione stessa.
- esempio: factorialrec

```
unsigned long long int factorialrec(int n ) {
  if(n>0) {
    return n*factorialrec(n-1);
  } else {
    return 1;
  }
}
```

Compilazione di funzioni private

Creare funz.c e funz.h,

includere funz.h sia in prog.c che in funz.c,

compilare separatamente programma e funz col comando

```
gcc -c prog.c
gcc -c funz.c
```

linkarli insieme col comando

```
gcc prog.o funz.o -o eseguibile
    (o anche direttamente:
gcc prog.c funz.c -o eseguibile)
```

Librerie private

- Tutte le funzioni da voi create possono essere raccolte in una libreria: in questo caso dopo

```
gcc -c funz.c
```

si usa il comando `ar` per *archiviarle* in una libreria, ad esempio

```
ar –r libmy.a funz.o
```

e il comando `ranlib` per aggiornare l'indice delle funzioni in libreria

```
ranlib libmy.a
```

- Un programma può caricarsi le funzioni da libreria mediante il comando

```
gcc  prog.c libmy.a -o eseguibile
```

- o piu' in generale

```
gcc  prog.c -Lmylibdir -lmy -o eseguibile
```

dove mylibdir e' l'indirizzo della directory contenente la libreria e my si riferisce a libmy.a (nel caso della libm.a: -L si omette perche' la libreria si trova /usr/lib che e' un posto standard)

parametri delle funzioni

- gli argomenti in C vengono passati *by value* dalla funzione chiamante alla funzione chiamata
- la lista degli argomenti viene vista come una lista di espressioni valutate al momento della chiamata, convertite nei tipi dichiarati nel prototipo. I valori risultanti vengono passati come input alla funzione chiamata
- la funzione chiamata ha quindi accesso ai valori dei parametri di ingresso, che vengono copiati in un'altra area di memoria, e non ai loro indirizzi (passaggio *by reference*).

parametri delle funzioni

- NB non esiste alcun legame tra i nomi delle variabili usati nella definizione della funzione (parametri formali) ed i nomi usati nella chiamata alla funzione stessa
- E' possibile passare dati *by reference* fornendo alla funzione chiamata la posizione di memoria in cui il dato si trova al momento della chiamata. L'indirizzo verra' passato *by value* ma permettera' l'accesso *by reference* al dato in esso contenuto.
- una funzione ritorna un unico risultato, mediante l'istruzione return. Vedremo che si possono definire dei nuovi tipi (strutture) e ritornare variabili strutturate contenenti un maggior umero di informazioni.