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 // lo vedremo nel prossimo corso!
- const altro_parametro

Esempio: elevazione a potenza

- Per elevare un numero x alla potenza y ci basta sapere che nelle librerie del C (che carichiamo automaticamente col comando g++) 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* accetta 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:
  - cout << fabs(-1) << endl;
  - cout << sqrt(2.0) << endl;
  - cout << pow(2.5,6) << endl;

Informazioni fornite dal prototipo:

- il tipo (classe) di oggetto ritornato dalla funzione
- il nome della funzione
- il numero di argomenti da usare nella chiamata
- il tipo (classe) degli argomenti

Moduli

- Un modulo è una raccolta di cose collegate tra di loro, quali funzioni, costanti e classi.
- Ad esempio il modulo math rappresenta una raccolta 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 (implicitamente se si tratta di librerie note al compilatore, esplicitamente in caso di librerie private)

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
- Il C++ costringe il programmatore a scrivere codice modulare, altri linguaggi no, ma è comunque sempre buona norma farlo!

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 files.
- Assumiamo di avere il main in progr.cc. Se questo fa uso di funzioni che abbiamo implementato nel file funz.cc è necessario fornire i prototipi delle funzioni in un file funz.h, che dovrà essere incluso all'inizio di prog.cc.

Esempio: riordino di una lista

- Consideriamo il seguente problema:

data una lista di numeri

89,93,75,36,56,78,67,23,59,51

vogliamo organizzarli in ordine crescente

Senza usare funzioni

```
int main( ) {
   int A[10]= {89,93,75,36,56,78,67,23,59,51};
   for(int j=0; j<9; j++) {
         for(int k=j+1; k<10; k++) {
                  if(A[j]>A[k]) {
                           int B=A[j];
                           A[j]=A[k];
                           A[k]=B;
                  }
         }
   }
return 1; }
```

Si ottiene

89,93,75,36,56,78,67,23,59,51 // j=0
75,93,89,36,56,78,67,23,59,51 // k=2
36,93,89,75,56,78,67,23,59,51 // k=3
23,93,89,75,56,78,67,36,59,51 // k=7
23,89,93,75,56,78,67,36,59,51 // j=1
23,75,93,89,56,78,67,36,59,51 // k=3
23,56,93,89,75,78,67,36,59,51 // k=4
23,36,93,89,75,78,67,56,59,51 // k=7
E così via, fino ad arrivare a
23,36,51,56,59,67,75,78,89,93

modularizzazione

- Nel riordinare la lista abbiamo cercato il numero più piccolo tra i 10 elementi e lo abbiamo collocato nell'elemento 0, poi il numero più piccolo tra i succesivi 9 elementi e lo abbiamo collocato nell'elemento 1... Di fatto abbiamo ripetuto per ogni elemento del nuovo vettore la ricerca del più piccolo tra i rimanenti elementi.
- Introduciamo allora una funzione che ci consenta di determinare quale degli elementi di un vettore sia il più piccolo e che abbia come argomenti il vettore stesso, e gli indici del primo e dell'ultimo elemento tra i quali cercare il minimo.

Introducendo la funzione minList

```
#include "funz.h"  // gli header privati si includono tra " "
                   // funz.h contiene il prototipo di minList
int main( ) {
    int A[10]= {89,93,75,36,56,78,67,23,59,51};
    for(int j=0; j<10; j++) {
          int minimo=minList(A, j,9);
           int B=A[minimo];
           A[minimo]=A[j];
           A[j]=B;
    }
 return 1;
 }
```

funz.h

```
#ifndef FUNZ_H
#define FUNZ_H

 // questa funzione restituisce l'indice dell'intero più
 // piccolo tra gli elementi del vettore lista
 // compresi tra l'elemento con indice primo e l'elemento
 // con indice ultimo.
// PRECONDIZIONE ultimo > primo
 int minList(int lista[  ],int primo, int ultimo);

#endif
```

funz.cc

```
#include "funz.h"

int minList(int lista[  ],int primo,int ultimo) {
   int curMin = primo;
   for(int j = primo+1; j<=ultimo; j++)
        if(lista[j] < lista[curMin]) curMin = j;
   return curMin;
 }
```

assert

void assert(int espressione)

il cui prototipo si trova in assert.h, è una funzione che valuta l'espressione che le viene data come argomento e se questa è falsa ferma il programma.

La si usa per proteggere le funzioni da situazioni che possano dar luogo ad operazioni illegali: nel nostro caso vogliamo protegger minList, che esegue un ciclo dall'indice primo all'indice ultimo dalla situazione in cui primo>ultimo.

funz.cc

```
#include "funz.h"
#include <assert.h>

int minList(int lista[  ],int primo,int ultimo) {
   int curMin = primo;
   assert(ultimo > primo);
   for(int j = primo+1; j<=ultimo; j++)
        if(lista[j] < lista[curMin]) curMin = j;
   return curMin;
 }
```

Si ottiene

89,93,75,36,56,78,67,23,59,51 // A[10] iniziale
89,93,75,36,56,78,67,23,59,51 // l'elemento più piccolo tra A[0] e A[9]
23,93,75,36,56,78,67,89,59,51 // lo si mette in A[0]
23,93,75,36,56,78,67,89,59,51 // si cerca tra A[1] e A[9]
23,36,75,93,56,78,67,89,59,51 // si cerca tra A[2] e A[9]
23,36,51,93,56,78,67,89,59,75
23,36,51,56,93,78,67,89,59,75
23,36,51,56,59,78,67,89,93,75
23,36,51,56,59,67,78,89,93,75
23,36,51,56,59,67,75,78,93,89
23,36,51,56,59,67,75,78,93,89
23,36,51,56,59,67,75,78,89,93

modularizzazione ulteriore

- Osservando che, una volta trovato il minimo abbiamo sempre bisogno di effettuare uno scambio tra due elementi potremmo introdurre la funzione

  void swap(double & a, double & b)

  che dati due numeri a e b restituisce a=b(iniziale) e b=a(iniziale). Il significato dei double & sarà spiegato nel prossimo corso, ma per il momento

  è importante osservare che i parametri passati ad una funzione non possono essere modificati dalla funzione stessa a meno di non essere "passati by reference" (double & ...).

swap (in funz.h)

```
void swap(double & a, double & b);
```

swap (in funz.cc)

```
void swap(double & a, double & b){
   double temp = a;
   a = b;
   b = temp;
   }
```

Introducendo la funzione swap

```
#include "funz.h"  // gli header privati si includono tra " "
                   // contiene il prototipo di minList e di swap
int main( ) {
    int A[10]= {89,93,75,36,56,78,67,23,59,51};
    for(int j=0; j<10; j++) {
          swap(A[minList(A, j,9)],A[j]);
    }
 return 1;
 }
```

Compilazione di funzioni private

- Due possibilità
  1. Inserire tutte le funzioni all'interno di prog.cc, ponendo i loro prototipi all'inizio del file e compilare normalmente prog.cc (sconsigliato: non vi consente di riutilizzare le funzioni in un altro programma)
  2. Creare funz.cc e funz.h, includere funz.h sia in prog.cc che in funz.cc, compilare separatamente programma e funz col comando

     g++ -c prog.cc

     g++ -c funz.cc

     e *linkarli* insieme col comando

     g++ -o eseguibile programma.o funz.o

Librerie private

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

  g++ -c funz.cc

  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

  g++ -o eseguibile prog.cc libmy.a