Corso “Programmazione 1”
Capitolo 05: Le Funzioni

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Giovanni De Toni - giovanni.detoni@unitn.it
Stefano Berlato - stefano.berlato-1@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 6 ottobre 2021

Terms of Use and Copyright

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Concetto di funzione

In un programma è sempre opportuno e conveniente strutturare il codice raggruppandone delle sue parti in sotto-programmi autonomi, detti funzioni, che vengono eseguite in ogni punto in cui è richiesto.

- L'organizzazione in funzioni ha moltissimi vantaggi:
  - Miglior strutturazione e organizzazione del codice
  - Maggior leggibilità del codice
  - Maggior mantenibilità del codice
  - Riutilizzo di sotto-parti di uno stesso programma più volte
  - Condivisioni di sotto-programmi tra programmi distinti
  - Utilizzo di codice fatto da altri/librerie
  - Sviluppo di un programma in parallelo, tra più autori
  - ...
- In un programma C++ è possibile definire e chiamare funzioni
- È possibile anche chiamare funzioni definite altrove
  - funzioni definite in altri file
  - funzioni di libreria

Funzioni di libreria

- Una funzione è un sotto-programma che può essere utilizzato ripetutamente in un programma, o in programmi diversi
- Una libreria è un insieme di funzioni precompilate.
- Alcune librerie C++ sono disponibili in tutte le implementazioni e con le stesse funzioni (ad es. `cmath`)
- Una libreria è formata da una coppia di file:
  - un file di intestazione (header) contenente le dichiarazioni dei sotto-programmi stessi
  - un file oggetto contenente le funzioni compilate
- Per utilizzare in un programma le funzioni in una libreria bisogna:
  - includere il file di intestazione della libreria con la direttiva **`#include`** `<nomelibreria>`
  - in alcuni casi, indicare al linker il file contenente le funzioni compilate della libreria
  - introdurre nel programma chiamate alle funzioni della libreria

Alcuni esempi di funzioni di libreria I

- Libreria `<cmath>`: funzioni matematiche (da **`double`** a **`double`**)
  - `fabs(x)`: valore assoluto di tipo float
  - `sqrt(x)`: radice quadrata di x
  - `pow(x,y)`: eleva x alla potenza di y
  - `exp(x)`: eleva e alla potenza di x
  - `log(x)`: logaritmo naturale di x
  - `log10(x)`: logaritmo in base 10 di x
  - `sin(x)` e `asin(x)`: seno e arcoseno trigonometrico
  - `cos(x)` e `acos(x)`: coseno e arcocoseno trigonometrico
  - `tan(x)` e `atan(x)`: tangente e arcotangente trig.
  - ...
- possono essere usate con tutti gli altri tipi numerici tramite conversione implicita o esplicita

Alcuni esempi di funzioni di libreria II

- Libreria `<cctype>`, funzioni di riconoscimento (da **`char`** a **`bool`**):
  - `isalnum(c)`: carattere alfabetico o cifra decimale
  - `isalpha(c)`: carattere alfabetico
  - `iscntrl(c)`: carattere di controllo
  - `isdigit(c)`: cifra decimale
  - `isgraph(c)`: carattere grafico, diverso da spazio
  - `islower(c)`: lettera minuscola
  - `isprint(c)`: carattere stampabile, anche spazio
  - `isspace(c)`: spazio,salto pagina,nuova riga o tab.
  - `isupper(c)`: lettera maiuscola
  - `isxdigit(c)`: cifra esadecimale
  - ...
- Libreria `<cctype>`, funzioni di conversione (da **`char`** a **`char`**):
  - `tolower(c)`: se `c` è una lettera maiuscola restituisce la corrispondente lettera minuscola, altrimenti restituisce `c`
  - `toupper(c)`: come sopra ma in maiuscolo
  - ...

Esempio di uso di funzioni di libreria

```
#include <cmath>
(...)
  for (float i=1.0; i<=MAX; i+=1.0)
    cout <<  log(i)/log(2.0)  << endl; // log2(i)
(...)
// dallo header della libreria cmath:
double log(double x);
```

- alla chiamata `log(i)`:
  - Il programma trasferisce il controllo dal codice di `main` al codice di `log` in `cmath`, e lo riprende al termine della funzione
  - il valore di `i` viene valutato e passato in input alla funzione `log`
  - `log(i)` viene valutata al valore restituito dalla computazione della funzione `log` con il valore `i` in input

Esempio di cui sopra (esteso):

{ `FUNCTIONS/tavola_logaritmi.cc` }

Funzioni: Dichiarazione, Definizione e Chiamata

- Definizione:
  - Sintassi: `tipo id(tipo1 id1, ... , tipoN idN) {...}`
  - Esempio: `double pow(double x, double y) {...}`
  - `id1,...,idN` sono i parametri formali (sempre presenti) della funzione
- Dichiarazione:
  - Sintassi: `tipo id(tipo1 [id1], ... , tipoN [idN]);`
  - Esempio: `double pow(double, double e);`
  - Serve per "richiamare" una definizione fatta altrove, e consentirne l'uso!
  - Nota: `id1,...,idN` sono opzionali!
- Chiamata:
  - Sintassi: `id (exp1, ..., expN)`
  - Esempio: `x = pow(2.0*y,3.0);`
  - `exp1, ..., expN` sono i parametri attuali della chiamata

Nota

I parametri attuali `exp1, ..., expN` della chiamata devono essere compatibili per numero, ordine e rispettivamente per tipo ai corrispondenti parametri formali!

L'istruzione `return`

- Il corpo di una funzione può contenere una o più istruzioni **`return`**
  - Sintassi: **`return`** `expression;`
  - Esempio: **`return`** `3*x;`
- `expression` deve essere compatibile con il tipo restituito dalla funzione
- L'esecuzione dell'istruzione **`return`**:
  - fa terminare la funzione
  - fa sì che il valore della chiamata alla funzione sia il valore dell'espressione `expression` (con conversione implicita se di tipo diverso)

Nota

È buona prassi che una funzione contenga un'unica istruzione **`return`**!

Esempio: la funzione `mcd`

Esempio di funzione:
{ `FUNCTIONS/mcd.cc` }

La chiamata `mcd(n1,n2)` viene eseguita nel modo seguente:

(i) vengono calcolati i valori dei parametri attuali `n1` e `n2` (l'ordine non è specificato)

(ii) i valori vengono copiati, nell'ordine, nei parametri formali `a` e `b` (chiamata per valore)

(iii) viene eseguita la funzione e modificati i valori di `a` e `b` e della variabile locale resto (`n1` e `n2` rimangono con il loro valore originale)

(iv) la funzione `mcd` restituisce al programma chiamante il valore dell'espressione che appare nell'istruzione **`return`**

```
int mcd(int  a, int  b){
  int resto;
  while(b!=0) {
     resto = a%b;
     a = b;
     b = resto;
   }
   return a;
}
```

Esempi

- Chiamate miste a funzioni definite e di libreria:
  { `FUNCTIONS/mylog10.cc` }
- funzione fattoriale:
  { `FUNCTIONS/fact.cc` }
- ...con dichiarazione (header):
  { `FUNCTIONS/fact1.cc` }
- ... con identificatore "fattoriale" locale e globale:
  { `FUNCTIONS/fact2.cc` }
- ... con parametro formale stesso nome di parametro attuale:
  { `FUNCTIONS/fact3.cc` }
- decomposto in più file:
  { `FUNCTIONS/fact4*.{cc|h}` }
- Esempio di funzione Booleana:
  { `FUNCTIONS/isprime.cc` }

Procedure (funzioni void)

In C++ c'è la possibilità di definire procedure, cioè funzioni che non ritornano esplicitamente valori (ovvero funzioni il cui valore di ritorno è di tipo **`void`**)

```
void pippo (int x) // definizione di funzione void
{...}
(...)
pippo(n*2); // chiamata di funzione void
```

Nelle funzioni **`void`**, l'espressione **`return`** può mancare, oppure apparire senza essere seguita da expressioni (termina la procedura).

- Es. di funzione **`void`**: stampa di una data:
  { FUNCTIONS/printdate.cc }
- Es. di funzione **`void`**: stampa di tutti i caratteri:
  { FUNCTIONS/printchartype.cc }
- Esempio di funzione senza argomenti:
  { FUNCTIONS/tiradadi.cc }

Return multipli in una funzione

- In una funzione è buona prassi evitare l'uso di **`return`** multipli (in particolare se usati come impliciti if-then-else)

```
int f (...) {
...
    return exp1;
...
    return expN;
}
```

$\Longrightarrow$

```
int f (...) {
int res;
...
    res =  exp1;
...
    res =  expN;
    return res;
}
```

```
if (...) {
    return exp1; }
... // altrimenti...
```

$\Longrightarrow$

```
if (...) {
    res = exp1; }
else { (...) }
```

- Es.: funzione `isprime` con return unico:
  { `FUNCTIONS/isprime_onereturn.cc` }

L'Istruzione `return` in un loop (salto implicito)

L'istruzione **`return`** termina direttamente il ciclo (e l'intera funzione)

- equivalente ad un **`break`**;
- Da evitare! $\Longrightarrow$ si può sempre fare modificando la condizione

```
int f () {
...
  while (...) {
    ...
    return ...; // --+
    ...           //   |
  }                     |
}       // <----------+
```

Conversione implicita dei parametri attuali

La chiamata (per valore) concettualmente analoga all'inizializzazione dei parametri formali

```
int f (int x, ...) {...}
...
... f(expr) ...
```

$\Longrightarrow$

```
...
int x = expr;
...
```

Nota importante

Nella chiamata a funzione in cui i parametri attuali siano di tipo diverso ma compatibile con quello dei rispettivi parametri formali, viene fatta una conversione implicita di tipo (con tutte le possibile problematiche ad essa associate)

- Regole analoghe a quelle dell'inizializzazione/assegnazione
- Es:

```
pow(2,4)        //conv. implicita da int a double
mcd(54.0,30.5) //conv. implicita da double a int
```

Ordine di valutazione di un'espressione II

- In C++ *non* è specificato l'ordine di valutazione degli operandi di ogni operatore, in particolare:
  - l'ordine di valutazione di sottoespressioni in un'espressione
  - l'ordine di valutazione degli argomenti di una funzione
- Es: nel valutare `f(expr1,expr2)`, non è specificato se `expr1` venga valutata prima di `expr2` o viceversa
- Problematico quando sotto-espressioni contengono operatori con "side-effects" come gli operatori di incremento.
  Es: `x=pow(++i,++i); //undefined behavior`
  $\Longrightarrow$ evitare l'uso di operatori con side-effects in chiamate a funzioni

Per approfondimenti si veda ad esempio
`http://en.cppreference.com/w/cpp/language/eval_order`

Parametri e variabili locali

- Un parametro formale è una variabile cui viene associato il corrispondente parametro attuale ad ogni chiamata della funzione
  - [se non diversamente specificato] il valore del parametro attuale viene copiato nel parametro formale
    (passaggio di parametri per valore)
- Le variabili dichiarate all'interno di una funzione sono dette locali
  - appartengono solo alla funzione in cui sono dichiarate
  - sono visibili solo all'interno della funzione
- Le variabili dichiarate all'esterno di funzioni sono dette globali
  - sono visibili all'interno di ogni funzione (se non mascherate da variabili locali con lo stesso nome)
- Esempio sull'ambito di parametri e variabili locali:
  { `FUNCTIONS/scope.cc` }

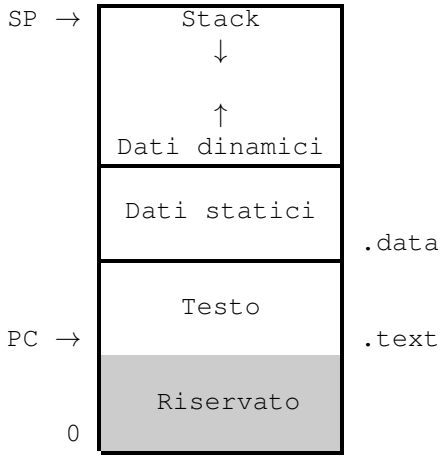
Durata di parametri e variabili locali

- I parametri formali e le variabili locali “esistono” (hanno uno spazio di memoria a loro riservato) solo durante l’esecuzione della rispettiva funzione
  (i) All’atto della chiamata viene riservata loro un’area di memoria
  (ii) Vengono utilizzati per le dovute elaborazioni
  (iii) Al termine della funzione la memoria da essi occupata viene resa disponibile

Modello di gestione della memoria per un programma

Area di memoria allocata ad un'esecuzione di un programma:

- Area programmi: destinata a contenere le istruzioni (in linguaggio macchina) del programma
- Area dati statici: destinata a contenere variabili globali o allocate staticamente e le costanti del programma.
- Area heap: destinata a contenere le variabili dinamiche (di dimensioni non prevedibili a tempo di compilazione) del programma.
- Area stack: destinata a contenere le variabili locali e i parametri formali delle funzioni del programma.



Modello a Stack

- I parametri formali e le variabili locali a una funzione sono memorizzate in un'area riservata della memoria, detta stack
- Modello di memoria concettualmente analogo a quello di una "pila" ("stack"):
  - quando una funzione viene chiamata, il blocco di memoria necessario per contenere i suoi parametri formali e variabili locali viene allocato "sopra" quello della funzione che la chiama
  - quando la funzione termina, tale blocco viene reso di nuovo disponibile
  - politica di gestione "last in first out" (LIFO)

Esempi

- esempio di funzioni che chiamano funzioni [D]:
  { `FUNCTIONS/comb.cc` }
- ...con dichiarazioni (headers):
  { `FUNCTIONS/comb2.cc` }
- ... tracciando gli indirizzi delle variabili e parametri:
  { `FUNCTIONS/comb2_track.cc` }
- chiamate annidate di funzioni [D]:
  { `FUNCTIONS/mymax.cc` }
- ... tracciando gli indirizzi (stack):
  { `FUNCTIONS/mymax_track.cc` }