Corso “Programmazione 1”
Capitolo 05: Le Funzioni

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Stefano Berlato - stefano.berlato-1@unitn.it
Andrea Mazzullo - andrea.mazzullo@unitn.it
Giovanna Varni - giovanna.varni@unitn.it
Matteo Franzil - matteo.franzil@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2023-2024
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 3 ottobre 2023

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
- `https://en.cppreference.com/w/cpp/header/cmath`

Alcuni esempi di funzioni di libreria II

- Nella libreria `<cstdlib>`: funzioni per numeri casuali
  - `abs(n)`: valore assoluto
  - `rand()`: numero pseudocasuale tra 0 e la costante `RAND_MAX`
  - `srand(n)`: inizializza la funzione rand
  - ...
- `https://en.cppreference.com/w/cpp/header/cstdlib`

Alcuni esempi di funzioni di libreria III

- 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
  - ...
- `https://en.cppreference.com/w/cpp/header/cctype`

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 { (...) }
return res;
```

- 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 ...; // --+
    ...           //   |
  }                     |
}         // <----------+
```

L'Istruzione `return` (errore tipico)

```
int f () {
  ...
  if (cond1) {
    ...
    return ...;
  }
  else {
    ... // senza return
  }
}
```

- Questo codice nel caso `cond1` sia falso, non esegue `return`!
- Il compilatore segnala che manca un `return` con un warning e non con un errore!
- È causa di errori, perchè non è definito che valore la funzione ritorna!
- Si ha però sensazione che tutto sia a posto (di solito viene ritornato il valore dell'ultima istruzione eseguita). Guardare con attenzione output del compilatore!

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` }

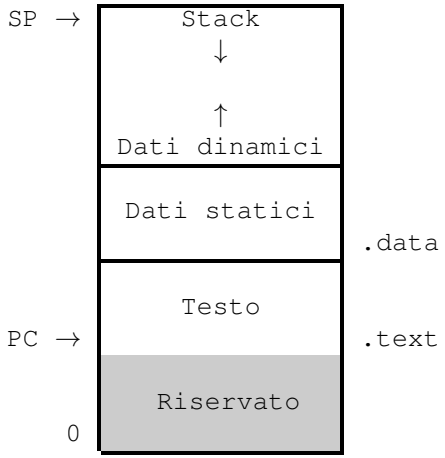
Durata di parametri e variabili locali

- 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 nell'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` }

Passaggio di parametri

In C++ esistono tre modalità passaggio di parametri a una funzione:

- per valore
- per riferimento
- per puntatore

(Spesso le ultime due sono confuse in letteratura, perché hanno finalità simili.)

Il passaggio di parametri per valore

- Definizione di parametri formali analoga a definizione di variabili
  - Sintassi lista dei parametri: `( tipo identificatore, ...)`
  - Es: `int fact(int n) {...}`
- Simile a definire una nuova variabile locale e assegnarle il valore dell'espressione del parametro attuale.
  - Es: `fact(3*x); //simile a: int n = 3*x;`
- Il parametro formale acquisisce il valore del parametro attuale
  - il parametro attuale può essere un'espressione senza indirizzo
  - può essere di tipo diverso compatibile $\Longrightarrow$ conversione implicita
- L'informazione viene (temporaneamente) duplicata
  $\Longrightarrow$ possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, il parametro attuale non viene modificato
  $\Longrightarrow$ passaggio di informazione solo dalla chiamante alla chiamata
- Tutti gli esempi di funzioni visti finora usano passaggio per valore:
  { `FUNCTIONS/...` }

Il passaggio di parametri per riferimento

- Definizione di parametri formali simile a definizione di riferimenti
  - Sintassi lista dei parametri: `( tipo & identificatore,...)`
  - Es: `int swap(int & n, int & m) {...}`
- Simile a definire un riferimento “locale” ad un’espressione dotata di indirizzo
  - Es: `swap(x,y); //simile a: int & n=x; int & m=y`
- Il parametro è un riferimento al parametro attuale
  - il parametro attuale deve essere un’espressione dotata di indirizzo
  - deve essere dello stesso tipo
- L’informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, modifico il parametro attuale
  $\Longrightarrow$ passaggio di informazione dal chiamante alla chiamata, ma ...
  anche dalla chiamata alla chiamante

Esempi

- passaggio per valore, errato:
  { `FUNCTIONS/scambia_err.cc` }
- passaggio per riferimento, corretto [D]:
  { `FUNCTIONS/scambia.cc` }
- passaggio per riferimento non ammesso, tipo diverso:
  { `FUNCTIONS/riferimento_err.cc` }
- problemi ad usare il riferimento quando non dovuto:
  { `FUNCTIONS/mcd_err.cc` }
- restituzione di due valori :
  { `FUNCTIONS/rectpolar.cc` }
- parametro come input e output di una funzione:
  { `FUNCTIONS/iva.cc` }

Passaggio di parametri per riferimento costante

- È possibile definire passaggi per riferimento in sola lettura (passaggio per riferimento costante)
  - Sintassi: `(const tipo & identificatore, ...)`
  - Es: `int fact(const int & n,...) {...}`
- Riferimento: l'informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Non permette di modificare `n`!
  - Es: `n = 5; //ERRORE!`
  - $\Longrightarrow$ passaggio di informazione solo dalla chiamante alla chiamata
  - $\Longrightarrow$ solo un input alla funzione
- Usato per passare in input alla funzione oggetti "grossi"
  - efficiente (no spreco di CPU e memoria)
  - evita errori
  - permette di individuare facilmente gli input della funzione
- Uso di riferimenti costanti:
  { `FUNCTIONS/usa_const.cc` }

Esempi (2)

- esempi per riferimento:
  { `FUNCTIONS/cipeciop.cc` }
- ...:
  { `FUNCTIONS/pippo.cc` }
- ...:
  { `FUNCTIONS/paperino.cc` }
- ...:
  { `FUNCTIONS/topolino.cc` }

Morale

Con i riferimenti è facile fare confusione!

Il passaggio di parametri per puntatore

- Definizione di parametri formali: puntatori passati per valore
  - Sintassi lista dei parametri: `( tipo * identificatore,...)`
  - Es: `int swap(int * pn, int * pm) {...}`
  - N.B.: nella chiamata, si passa l'indirizzo dell'oggetto passato
- Simile a definire un puntatore "locale" ad un'espressione dotata di indirizzo
  - Es: `swap(&x,&y);//simile a: int *pn=&x; int *pm=&y`
- Il parametro è un puntatore al(l'oggetto il cui indirizzo è dato dal) parametro attuale
  - che deve essere un'espressione dotata di indirizzo
  - che deve essere dello stesso tipo
- L'informazione non viene duplicata
  $\Longrightarrow$ evito possibile spreco di tempo CPU e memoria
- Se modifico il parametro formale, modifico il parametro attuale
  $\Longrightarrow$ passaggio di informazione anche dalla chiamata alla chiamante

$\Longrightarrow$ effetto simile al passaggio per riferimento (vedi C)

Esempi

- come scambia.cc, con passaggio per puntatore:
  { `FUNCTIONS/scambia_punt.cc` }
- come iva.cc, con passaggio per puntatore:
  { `FUNCTIONS/iva2.cc` }
- come paperino.cc, con passaggio per puntatore:
  { `FUNCTIONS/paperino2.cc` }

Passaggio per valore vs. p. per riferimento/puntatore

- Vantaggi del passaggio per riferimento/puntatore:
  - Minore carico di calcolo e di memoria
    (soprattutto con parametri di grosse dimensioni)
  - Permette di restituire informazione da chiamata a chiamante
- Svantaggi del passaggio per riferimento/puntatore:
  - Rischio di confusione nel codice (non si sa dove cambiano i valori)
  - Aliasing (entità con più di un nome)
  - Parametro formale e attuale esattamente dello stesso tipo
  - Si possono passare solo espressioni dotate di indirizzo

Nota

- alcuni linguaggi (es C) non ammettono passaggio per riferimento (solo per puntatore)

Esercizi Proposti

Esercizi su funzioni:
{ FUNCTIONS/ESERCIZI_PROPOSTI.txt }

Funzioni che restituiscono un riferimento

- Restituisce un riferimento ad un'espressione (con indirizzo)
  - l'espressione deve riferirsi ad un oggetto del chiamante (es. un parametro formale passato per riferimento, un elemento di un array)
  - deve essere dello stesso tipo
- La chiamata è un'espressione dotata di indirizzo!

```
int& max(int& x,int& y) //restituisce un riferimento
{return (x > y ? x : y);}//x,y riferimenti a oggetti
                         //non locali
(...)
int m=44, n=22;
max(m,n) = 55;     //cambia il valore di m da 44 a 55
```

Esempio di cui sopra esteso:
{ FUNCTIONS/restituzione_riferimento.cc }

Sovrapposizione di parametri (overloading)

- In C++ è possibile dare lo stesso nome a funzioni diverse, purché con liste di parametri diverse, per numero e/o per tipo
- Il compilatore “riconosce” la giusta funzione per ogni chiamata.
- In caso di ambiguità, il compilatore produce un errore
- Conversioni implicite ammissibili, purché non causino ambiguità

```
int max(int, int) {...};
int max(int, int, int) {...};
double max(double, double) {...};
(...)
  cout << max(99,77) << "␣" << max(55,66,33) << "␣"
       << max(3.4,7.2) << endl;
  cout << max(3,3.1) << endl; // errore: AMBIGUA
```

Esempio di cui sopra esteso:

{ FUNCTIONS/overloading.cc }

Funzioni con argomenti di default (cenni)

- In C++ è possibile fornire parametri opzionali, con valori di default
  - permette chiamate con liste di parametri attuali ridotte
  - i parametri opzionali devono essere gli ultimi della lista
  - il match viene effettuato da sinistra a destra

```
double p(double, double, double =0, double =0, double =0);

  cout << p(x, 7) << endl;
  cout << p(x, 7, 6) << endl;
  cout << p(x, 7, 6, 5) << endl;
  cout << p(x, 7, 6, 5, 4) << endl;

double p(double x, double a0, double a1, double a2, double a3)
{ return a0 + (a1 + (a2 + a3*x)*x)*x; }
```

Esempio di cui sopra esteso:
{ FUNCTIONS/defaultvalues.cc }

Funzioni ricorsive

- In C++ una funzione può invocare se stessa (funzione ricorsiva)
- ... o due o più funzioni possono chiamarsi a vicenda (funzioni mutualmente ricorsive)
- Formulare alcuni problemi in maniera ricorsiva risulta naturale:
  - il fattoriale: $0! \stackrel{\text{def}}{=} 1; n! \stackrel{\text{def}}{=} n \cdot (n-1)!$
  - pari/dispari: $even(n) \iff odd(n-1)$; $odd(n) \iff even(n-1)$;
  - espressioni: $somma \stackrel{\text{def}}{=} numero; somma \stackrel{\text{def}}{=} (somma + somma)$
- Due componenti:
  - una o più condizioni di terminazione
  - una o più chiamate ricorsive
- Intrinseco rischio di produrre sequenze infinite
  - Analoghe considerazioni rispetto ai cicli
- Alcune “insidie” computazionali
  - Es: funzione di Fibonacci: $f_0 \stackrel{\text{def}}{=} 1; f_1 \stackrel{\text{def}}{=} 1; f_n \stackrel{\text{def}}{=} f_{n-1} + f_{n-2}$

Ricorsione fortemente collegata al principio di induzione matematico.

Esempi

- fattoriale:
  { `FUNZIONI_RICORSIVE/fact_nocomment.cc` }
- ..., con chiamate tracciate:
  { `FUNZIONI_RICORSIVE/fact.cc` }
- ..., errore (loop infinito) :
  { `FUNZIONI_RICORSIVE/fact_infloop.cc` }
- ..., stack tracciato :
  { `FUNZIONI_RICORSIVE/fact_stack.cc` }
- funzioni mutualmente ricorsive:
  { `FUNZIONI_RICORSIVE/pariDispari.cc` }
- Fibonacci:
  { `FUNZIONI_RICORSIVE/fibonacci_nocomment.cc` }
- ..., con chiamate tracciate:
  { `FUNZIONI_RICORSIVE/fibonacci.cc` }
- versione iterativa:
  { `FUNZIONI_RICORSIVE/fibonacci_iterativa.cc` }

Nota sulla ricorsione

La realizzazione ricorsiva di una funzione può richiedere due funzioni:

- una funzione ausiliaria ricorsiva, con un parametro di ricorsione aggiuntivo (simile a contatore in loop)
- una funzione principale (wrapper) che chiama la funzione ricorsiva con un valore base del parametro di ricorsione
- situazione molto frequente nell'uso di array (prossimo capitolo)

- Esempio di funz. ricorsiva che necessita wrapper:
  { `FUNZIONI_RICORSIVE/stampanumeri.cc` }
- ... variante 1:
  { `FUNZIONI_RICORSIVE/stampanumeri1.cc` }
- ... variante 2:
  { `FUNZIONI_RICORSIVE/stampanumeri2.cc` }
- analoga variante del fattoriale, con wrapper:
  { `FUNZIONI_RICORSIVE/fact_rec1.cc` }

Ricorsione vs. Iterazione

- Ricorsione spesso più naturale, semplice ed elegante
- Efficienza della ricorsione critica:
  - Attenzione a chiamate identiche in rami diversi! (es. Fibonacci)
    $\Longrightarrow$ rischio esplosione combinatoria
  - Dimensione dello stack dipende dalla profondità di ricorsione
    $\Longrightarrow$ notevole overhead e spreco di memoria
    $\Longrightarrow$ quando possibile, tipicamente iterazione più efficiente
    $\Longrightarrow$ passando oggetti "grossi", è indispensabile usare passaggio per riferimento o puntatore
- Molte funzioni ricorsive possono essere riscritte in forma iterativa:
  - tail recursion: una chiamata ricorsiva, operata come ultimo passo
    - Es: somma, pari/dispari, ...
  - in generale, quando non comporta una "biforcazione"
    - Es: fattoriale, Fibonacci, ...
  - `g++ -O2` effettua una conversione da tail-recursive in iterative

Da ricorsione in coda a iterazione (caso void)

```
void F(int x,...) {
 if (CasoBase(x,...))
  {IstrBase(...);}
 else {
  Istr(...);
  x=agg(x,...);
  F(x,...);
}}
```

$\Longleftrightarrow$

```
void F(int x,...) {
 while (!CasoBase(x,...)){
  Istr(...);
  x=agg(x,...);
 }
 {IstrBase(...);}
}
```

- Esempio funzione void tail-recursive :
  { FUNZIONI_RICORSIVE/stampanumeri3.cc }
- ... corrispondente versione iterativa :
  { FUNZIONI_RICORSIVE/stampanumeri3_while.cc }

Da ricorsione in coda a iterazione (caso generale)

```
type F(int x,...) {
 if (CasoBase(x,...))
  res = IstrBase(...);
 else {
  Istr(...);
  x=agg(x,...);
  res = F(x,...);
 }
 return res;
}
```

$\Longleftrightarrow$

```
type F(int x,...) {
 while (!CasoBase(x,...)){
  Istr(...);
  x=agg(x,...);
 }
 res = IstrBase(...);
 return res;
}
```

- Esempio funzione tail-recursive:
  { FUNZIONI_RICORSIVE/sum.cc }
- ... corrispondente versione iterativa:
  { FUNZIONI_RICORSIVE/sum_while.cc }
- Compilazione di funzioni tail-recursive in iterative:
  { FUNZIONI_RICORSIVE/tailrecursive-comp.cc }

Ricorsione vs. Iterazione II

- ...
- Talvolta non è agevole riscrivere la ricorsione in forma iterativa
  - funzioni non-tail recursive, chiamate multiple
  - Es: manipolazione di espressioni

  Esempio di gestione di espressioni:
  { `FUNZIONI_RICORSIVE/espressione.cc` }
- In generale, convertire una funzione ricorsiva in iterativa richiede l'uso di uno stack

Esercizi Proposti

Esercizi su funzioni ricorsive:
{ `FUNZIONI_RICORSIVE/ESERCIZI_PROPOSTI.txt` }