PROGRAMMAZIONE AVANZATA

1 Funzioni e strutture

- Sommario
- Principio di astrazione funzionale
- Soluzione (imperfetta)
- Soluzione con astrazione funzionale
- Sviluppo di una libreria di funzioni
- Strutture per dati composti
- Creazione da numeri interi
- Creazione da numeri in virgola mobile
- Conversioni
- Operazioni
- Esercizi

Sommario

Ripasso

- Strutture e funzioni
- Principio di **astrazione funzionale**
- Realizzazione di una semplice **libreria di funzioni**

Nella prossima lezione

- Usiamo la libreria come punto di partenza per introdurre i concetti base della programmazione a oggetti
- Trasformiamo la libreria in una **classe**

Principio di astrazione funzionale

- Se lo **stesso codice** serve in più parti del programma, si scrive una **funzione** che lo realizza una volta sola e si **invoca**/**applica** la funzione ovunque è necessario.
- Gli **argomenti** della funzione astraggono i dati che possono variare da un'invocazione all'altra.

Esempio

Calcolare il coefficiente binomiale

$$\binom{n}{k} \stackrel{\text{def}}{=} \frac{n!}{k!(n-k)!}$$

dati n e k tali che $0 \leq k \leq n$.

Soluzione (imperfetta)

```
int bin(int n, int k) {
  int a = 1;                              // calcolo n!
  for (int i = 1; i ⩽ n; i++)
    a = a * i;
  int b = 1;                              // calcolo k!
  for (int i = 1; i ⩽ k; i++)
    b = b * i;
  int c = 1;                              // calcolo (n - k)!
  for (int i = 1; i ⩽ n - k; i++)
    c = c * i;
  return a / (b * c);
}
```

- devo calcolare il fattoriale di **tre numeri diversi**
- il codice per il calcolo del fattoriale è triplicato

Soluzione con astrazione funzionale

```
int fact(int n) {
  int r = 1;
  for (int i = 2; i ⩽ n; i++) r = r * i;
  return r;
}

int bin(int n, int k) {
  return fact(n) / (fact(n - k) * fact(k));
}
```

- La funzione `fact` calcola il fattoriale di un generico `n`
- Scrivo `fact` una volta e in `bin` la applico tre volte

Sviluppo di una libreria di funzioni

Obiettivo

- Il C++ ha tipi primitivi `int` (numeri interi) e `double` (numeri in "virgola mobile")
- Vogliamo definire un nuovo tipo `rat` (numeri razionali)

Rappresentazione di numeri razionali

- numero razionale = rapporto di 2 numeri interi (numeratore, denominatore)
- Vincoli: denominatore strettamente positivo

Operazioni su numeri razionali

- **Creazione** da `int`, da `double`, da coppia di `int`
- **Conversione** a `int`, a `double`
- Operazioni **aritmetiche** somma, negazione, moltiplicazione, reciproco, ...
- **Stampa** sul terminale
- **Riduzione** ai minimi termini

Strutture per dati composti

Rappresentazione

```
struct rat {
  int num;
  int den;
};
```

- Introduce un **nuovo tipo di dato** `rat`
- Ogni valore di tipo `rat` comprende due **campi** chiamati `num` e `den`

Inizializzazione

```
rat r = { 1, 2 };
```

- Definisce `r` come un valore di tipo `rat` in cui `num = 1` e `den = 2`

Lettura e scrittura

```
r.num = 3;                          // scrive campo num
std::cout << r.den << std::endl;    // legge campo den
```

Creazione da numeri interi

```
rat rational(int a, int b = 1) {
  if (b > 0) {
    rat r = { a, b };
    return r;
  } else if (b < 0) {
    rat r = { -a, -b };
    return r;
  } else throw std::domain_error("division by zero");
}
```

- `b = 1` specifica il **valore di default** per l'argomento opzionale
- `throw` serve per "lanciare un'eccezione" (segnala anomalia)
- `std::domain_error` è il tipo delle anomalie che riguardano il dominio delle funzioni

Esempi

```
rat a = rational(1, 2);          // crea 1/2
rat b = rational(1);             // crea 1/1
```

Creazione da numeri in virgola mobile

```
#include <cmath>

...

rat rational(double a, int n) {
  int m = std::pow(10, n);
  return rational((int) (a * m), m);
}
```

- `cmath` definisce funzioni matematiche di uso comune (`pow`, `exp`, `sin`, ...)
- `(int)` è un **cast**: converte il valore di un'espressione al tipo indicato
- n = numero di cifre dopo la virgola, $m = 10^n$

Overloading

- Ci possono essere più definizioni di funzioni con lo **stesso nome**
- Il compilatore sceglie la funzione in base a **numero/tipo** degli argomenti

```
rat e = rational(std::exp(1), 5); // crea 271828 / 100000
```

Conversioni

```
int to_int(const rat& r) {
  return r.num / r.den;
}

double to_double(const rat& r) {
  return (double) r.num / r.den;
}
```

- Il tipo `rat&` significa **riferimento** a un valore di tipo `rat`
- Serve per **evitare la copia** del valore (maggior efficienza)
- Il qualificatore `const` significa che la funzione **non modifica** l'argomento
- Il cast `(double)` del numeratore è fondamentale per evitare il troncamento del risultato. Il denominatore viene **promosso** a `double`.

Esempi

```
std::cout << to_int(e) << std::endl;    // stampa 2
std::cout << to_double(e) << std::endl; // stampa 2.71828
```

Operazioni

```
rat add(const rat& a, const rat& b) {
  return rational(a.num * b.den + b.num * a.den,
                  a.den * b.den);
}

rat neg(const rat& a) {
  return rational(-a.num, a.den);
}
```

Ancora overloading

Si può fare overloading anche degli operatori predefiniti dal linguaggio:

```
rat operator+(const rat& a, const rat& b) {
  return add(a, b);
}
```

Così si può scrivere:

```
  rat r = rational(1, 2) + rational(3, 4);
```

Esercizi

1. Usando le funzioni `fact` e `bin` scrivere un programma `tartaglia.cc` che stampi una sezione finita (diciamo fino a $n = 10$) del triangolo di Tartaglia. Soluzione
2. Implementare funzioni per le seguenti operazioni su numeri razionali: sottrazione (`sub`), moltiplicazione (`mul`), reciproco (`recip`), divisione (`div`).
3. Implementare la funzione `print` per la stampa di numeri razionali nel formato n/d, facendo in modo che il numero stampato sia semplicemente n quando $d = 1$.
4. Implementare la funzione `reduce` per la riduzione ai minimi termini di un numero razionale. **Suggerimento**: usare una funzione ausiliaria `gcd` che calcola il massimo comun divisore di due numeri.
5. Implementare (possibilmente con una sola riga di codice) la funzione `pow_nn` per elevare un numero razionale a una potenza intera non negativa senza usare funzioni della libreria standard del C++.
6. Implementare (possibilmente con una sola riga di codice) la funzione `pow` per elevare un numero razionale a una potenza intera che può essere negativa.
7. Confrontare le proprie soluzioni con quelle fornite.