Gestione degli Errori

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Gli Errori Si Devono Evitare...

Consideriamo il costruttore dei razionali

```
class rational {
    unsigned int num;        //!< numeratore
    unsigned int denom;      //!< denominatore
...
    rational(const int num, const int denom)
        :  num{num}, denom{denom}
    {}
...
};
```

Gli Errori Si Devono Evitare... Ma Capitano

Consideriamo il costruttore dei razionali

```
class rational {
    unsigned int num;        //!< numeratore
    unsigned int denom;      //!< denominatore
...
    rational(const int num, const int denom)
        :  num{num}, denom{denom}
    {}
...
};
```

Cosa succede se `denom` è 0?

Le Eccezioni

Sono un meccanismo per **segnalare e recuperare** situazioni critiche

Se si verifica un evento critico, viene **lanciata un'eccezione**

Essa arresta l'esecuzione del codice in esecuzione

Il codice può **intercettare l'eccezione** per gestire l'evento

In C++ le eccezioni sono oggetti di classi derivate da `std::exception`

Alcune Eccezioni del C++ (in `std`)

- `bad_alloc` errore nell'allocazione di memoria
- `logic_error` identificabili prima dell'esecuzione
  - `domain_error` errore nel dominio della funzione
  - `out_of_range` l'argomento non sta nell'intervallo previsto
- `runtime_error` identificabili al runtime
  - `overflow_error` e `underflow_error` aritmetica intera
  - `range_error` aritmetica *floating point*
  - `system_error` dal sistema operativo
    - `ios_base::failure` errore dell'I/O

Gestire le Eccezioni in C++

Tramite in costrutto `try-catch`

```
#include <exception>    // per std::exception

std::string s{"Test"}
try {

    s.at(10) = 'x'; // solleva un'eccezione per la lunghezza di s

} catch (const std::exception& e) {
    std::cerr << "eccezione del tipo " << typeid(e).name()
              << " msg: \""<< e.what() << "\"" << std::endl;
}

std::cout << "Questa linea viene eseguita" << std::endl;
```

Gestire Diversi tipi di Eccezioni

```
try {

    // solleva un'eccezione

} catch (<TIPO_ECCEZIONE_A>) {     // il primo catch "compatibile" con il
                                   // tipo dell'eccezione sollevata
    // Gestisci tipo 1             // gestirà l'eccezione

} catch (<TIPO_ECCEZIONE_B>) {

    // Gestisci tipo 2

}
...
```

Propagazione di un Eccezione

Scorre la pila delle chiamate di funzioni fino a un `catch`

```
void test_throw(size_t count) {          // funzione ricorsiva
    if (count==0)  raise_exception();  // il caso base solleva un'eccezione

    test_throw(count-1);
}

try {
    test_throw(10);
} catch (...) {
    // gestisci l'eccezione
}
```

Eccezioni e `noexcept`

Sarebbe opportuno evitare di lanciare eccezioni perché inefficienti

Possiamo dichiarare che una funzione/metodo non lancerà eccezioni

```
int somma(int a, int b) noexcept     // noexcept dichiara che non sono
{                                    // previste eccezioni da somma
    return a+b;
}
```

Se `somma` solleva un'eccezione, il programma viene terminato

Eccezioni Personalizzate

`throw` può lanciare anche tipi base, e.g., `int`, `const char*`, ...

Se vogliamo eccezioni strutturate dobbiamo usare le classi

Classi derivate da `std::exception` che forniscono:

- costruttore di copia
- operatori di copia
- override del metodo `const char* what() const noexcept`

Esempio di Eccezione Personalizzata

```
class my_exception : public std::exception {
    std::string msg;    // il messaggio dell'eccezione

public:
    my_exception(const std::string &message)
        : msg(message)
    {}

    // costruttore e operatore di copia sono quelli standard

    const char* what() const noexcept {      // questo metodo è
        return msg.c_str();                  // necessario
    }
};
```