Gestione delle eccezioni

Programmazione di Sistema

A.A. 2017-18

Argomenti

- Eccezioni
- Strategie di gestione

Gestire le anomalie

- Non tutte le operazioni hanno sempre successo
  - Possono verificarsi errori e fallimenti di varia natura
  - Alcuni ascrivibili al comportamento dell'utente, altri al sistema
- Possibili fallimenti
  - Saturazione della memoria
  - Disco pieno
  - File assente
  - Accesso negato
  - Malfunzionamento hardware
  - Rete inaccessibile
  - …

Errori attesi

- Occorre verificare se siano o meno presenti
  - Gestendoli nel punto preciso in cui sono stati identificati
- Problemi
  - A volte il test viene omesso (int scanf(char*, ...))
  - Altre volte, l'errore viene rilevato in una funzione che non sa come trattarlo: occorre propagarlo all'indietro
- Problemi legati alla propagazione
  - Una funzione potrebbe ritornare già un valore
  - Un costruttore non può ritornare nulla
  - Cosa capita se il chiamante di una funzione non propaga un errore?

Errori non attesi

- Molto più difficili da gestire
  - Possono verificarsi pressoché ovunque
  - Il tentativo di gestirli rende praticamente illeggibile il codice
- È molto improbabile che sia possibile gestirli nel posto in cui si sono verificati
  - Se non si riesce ad allocare un oggetto interno ad un'implementazione mentre si invoca un metodo, cosa fare?
- Occorre un modo alternativo per poter strutturare il codice
  - Che permetta di fare fronte alle evenienze là dove si ha la possibilità di prendere una decisione opportuna

Eccezioni

- Meccanismo che permette di trasferire il flusso
  di esecuzione ad un punto precedente, dove si
  ha la possibilità di gestirlo
  - Saltando tutto il codice intermedio
  - Evitando il rischio di dimenticarsi di propagare una indicazione di errore
- Si notifica al sistema la presenza di un'eccezione creando un dato qualsiasi
  - E passandolo come valore alla parola chiave "throw"
- Quando il sistema esegue "throw"
  - Abbandona il normale flusso di elaborazione
  - E inizia la ricerca di una contromisura

Eccezioni

- Il tipo del dato passato a throw è arbitrario
  - Serve a descrivere quanto si è verificato
  - Il suo tipo viene utilizzato per scegliere quale contromisura adottare
- La libreria standard offre la classe std::exception
  - Può essere usata come base di una gerarchia di ereditarietà, per creare classi più specifiche
- Occorre documentare bene l'uso delle eccezioni
  - Evidenziandone la semantica

Eccezioni

- Se un’eccezione si verifica in un blocco try…
  - …o in un metodo chiamato dall’interno di un blocco try…
- …si contrae lo stack fino al blocco try incluso
  - Tutte le variabili locali vengono distrutte ed eliminate

Eccezioni

```
try {
  //le istruzioni eseguite qui possono
  //lanciare un'eccezione
}
catch (out_of_range& oor){
  //contromisura
  //per errore specifico
}
catch (exception& e)  {
  //contromisura generica
}
```


Eccezioni

- Le clausole “catch” effettuano un match basato sul tipo dell’eccezione
  - Se la classe dell’eccezione coincide o deriva da (is_a) quella indicata, il codice corrispondente è eseguito e l’eccezione rientra
- I blocchi catch sono esaminati nell’ordine in cui sono scritti
  - Occorre ordinarli dall’eccezione più specifica alla più generica
  - Se non c’è alcuna corrispondenza, l’eccezione rimane attiva e il programma ritorna alla funzione chiamante…

Eccezioni

- ...ed eventualmente al chiamante del chiamante...
  - Fino ad incontrare un blocco catch adatto...
  - ...o fino alla contrazione completa dello stack, con la terminazione del flusso di esecuzione

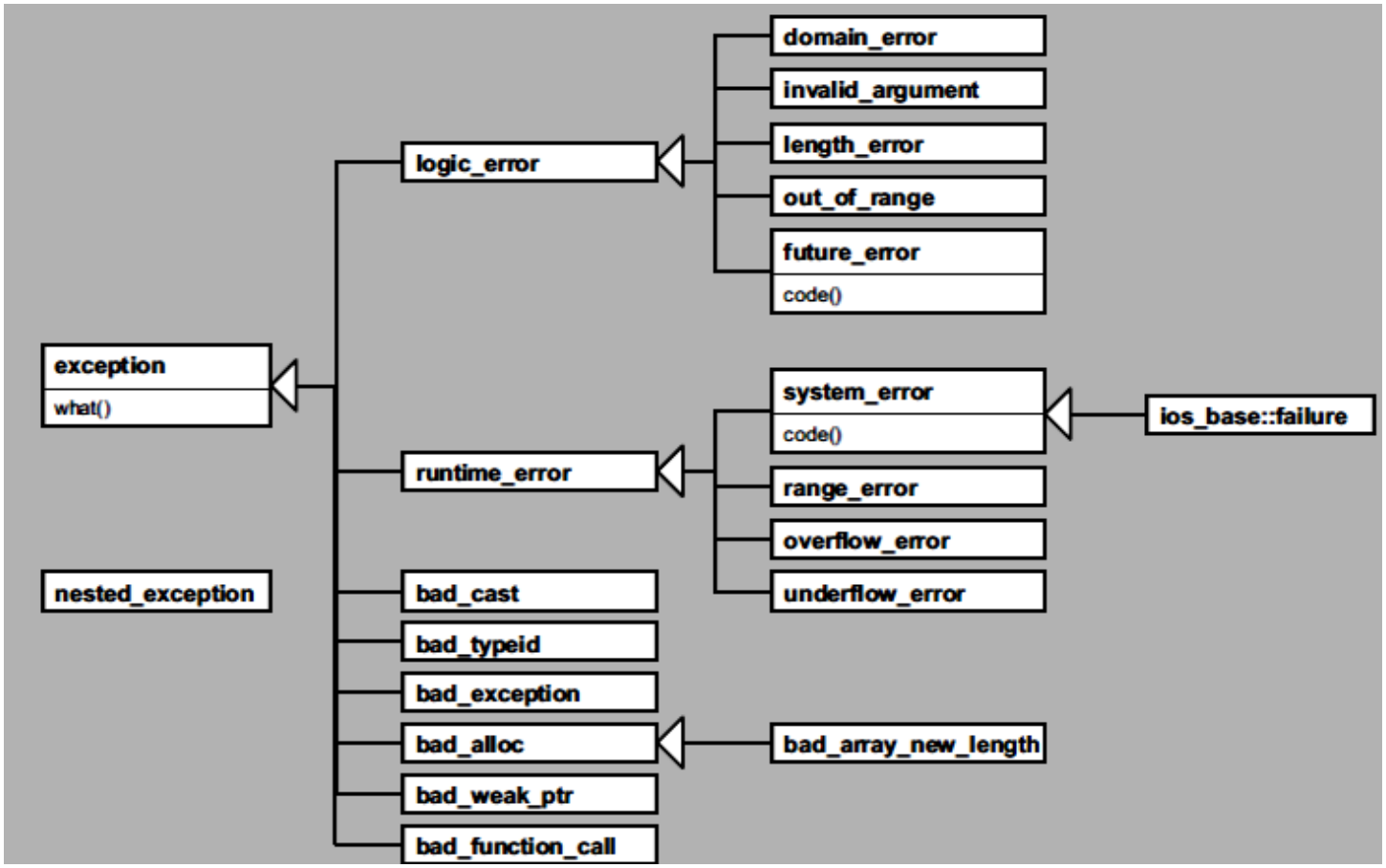
La classe std::exception

- Incapsula una stringa, che viene inizializzata nel costruttore
  - Si accede al suo contenuto con la funzione membro what()
  - Serve per documentare l'errore, NON per creare del codice che reagisca all'eccezione

Alcune classi derivate

- std::logic_error – incoerenza del codice
  - Principali sottoclassi:
    domain_error
    invalid_argument
    length_error
    out_of_range
- std::runtime_error – condizione inattesa
  - Sottoclassi:
    overflow_error,
    range_error
    underflow_error
    system_error

La classe std::exception



Dichiarare le eccezioni

- Le classi std::exception e suoi derivati sono dichiarate in diversi file
  - #include <exception>
  - std::exception e std::bad_exception
- #include <stdexcept>
  - Per la maggior parte delle classi relative agli errori logici e in fase di esecuzione
- #include <system_error>
  - Per gli errori di sistema (C++11)

Dichiarare le eccezioni

- #include <new>
  - Per le eccezioni relative alla mancanza di memoria
- #include <ios>
  - Per gli errori di I/O
- #include <future>
  - Per gli errori relativi all'esecuzione asincrona (C++11)
- #include <typeinfo>
  - Per le eccezioni legate ai cast dinamici o alla RTTI

Strategie di gestione

- Il blocco catch ha lo scopo di rimediare all'eccezione che si è verificata
  - In base al tipo di evento, occorre adottare una contromisura adeguata
- Strategie di gestione
  - Terminare, in modo ordinato, il programma
  - Ritentare l'esecuzione, evitando di entrare in un loop infinito
  - Registrare un messaggio nel log e rilanciare l 'eccezione

Terminare il programma

- Si procede a salvare lo stato del programma e rilasciare eventuali risorse globali
  - Azione estrema, di solito lasciata al livello più esterno (mai

```
try {
  //azioni a rischio
}
catch( ... )
{
  // eventuali azioni di salvataggio

  //codice di errore
  exit(-1);
}
```


Terminare il programma

- La sintassi “...” indica un’eccezione qualsiasi
- Permette l’ingresso nel blocco catch qualunque sia il tipo di dato lanciato (che però non è accessibile)

Ritentare l'esecuzione

- Strategia adatta quando il fallimento è dovuto a cause tem
  - Congestione di rete
  - Mancanza di risorse (disco, memoria) se l'applicazione può rilasciarne

```
int retry=2;
while (retry) {
  try {
    //azioni varie
    break; //termina
  }
  catch(exception& e) {
    ReleaseExtraRes();
    retry--;
    if (!retry) throw;
  }
}
```


Ritentare l'esecuzione

- L'istruzione "throw;" al termine del blocco catch rilancia l'eccezione catturata

Registrare un messaggio

- Si usa un meccanismo opportunamente predisposto per la registrazione degli errori
  - Come il flusso “std::cerr”
  - Il metodo what() di exception permette di accedere ad una descrizione dell’errore
  - L’istruzione throw

```
try {
        //...
} catch (exception& e) {
  logger::log(e.what());
  throw;
}
```

Cosa non fare

- Scrivere un blocco catch che non esegue nessuna strategia di riallineamento
  - E lascia che il programma, la cui esecuzione è parzialmente fallita, prosegua
- Stampare un messaggio di errore e poi continuare, in genere, non è una soluzione

```
try {
        //...
} catch (std::exception& e)  {
  std::cerr<<"Errore:"<<e.what()<<"\n";
}
```

Contrazione dello stack

- Il fatto che lo stack venga contratto in fase di lancio di un'eccezione è alla base di un pattern di programmazione tipico del C++
  - RAII – Resource Acquisition Is Initialization
- Aiuta a garantire che le risorse acquisite alla costruzione siano liberate
  - Può essere usato per eseguire azioni anche in presenza di eccezioni
  - Facendo attenzione a non sollevarne delle altre
- È l'unica alternativa al blocco "finally" di Java, che in C++ non esiste

RAII

```
class ActionTimer {
  long start;
  std::string msg;
public:
  ActionTimer(std::string tag):
    msg(tag){
    start=GetCurrentTime();
  }
  ~ActionTimer() {
    long t=GetCurrentTime()-start;
    std::cout<<msg<<":"<<t<<"\n";
  }
};
```

```
{
  ActionTimer at("b1");
  for(int i=0; i<8; i++){
   ActionTimer at2("b2");
    //do something...
    throw std::exception;
  }
}
```

I costi delle eccezioni

- Se nessuna eccezione viene lanciata, non ci sono sostanziali penalità
  - La definizione di un blocco try si limita ad inserire alcune informazioni nello stack
- Se viene lanciata un'eccezione il costo è abbastanza elevato
  - Almeno un ordine di grandezza maggiore dell'invocazione di una funzione …
  - … a cui si aggiunge il costo dell'esecuzione dei vari distruttori
- Non è un meccanismo conveniente per gestire logiche locali
  - Un costrutto "if" è molto meno oneroso

Spunti di riflessione

- Si scriva una classe dotata di costruttore e distruttore che incapsuli un valore
- Si verifichi con il debugger il funzion
  seguen

```
class MyClass;  //Da definirsi

int main(int argc, char** argv){
  MyClass c1(1);
  try {
    MyClass c2(2);
    throw "errore";
  } catch (…) {
    throw;
  }
}
```