Exceções em C++

Algoritmos e Estruturas de Dados

2020/2021

Mestrado Integrado em Engenharia Informática e Computação

Exceções

- Condições inesperadas surgem em qualquer programa
- O programador deve ser capaz de detetar erros e tratar esses erros

- Exceções

  Anomalias que ocorrem durante a execução de um programa. ex: perder a ligação à base de dados, *input* não esperado, ...

  – Se alguma das exceções não é tratada, o programa termina!

- Deteção de exceção e tratamento de exceção são partes independentes
  – deteção de exceção: *throw*, termina o processamento da função atual e procura *handler*
  – tratamento de exceção: *try* e *catch* (*catch* é denominado *handler*)

Exceções

- Tratamento de exceções
  - maneira de transferir controlo e informação de um ponto na execução do programa, para uma rotina de tratamento da exceção (*handler*) associada a um ponto anterior na execução (retorno automático de vários níveis)

- **`throw`** `objeto;`
  - lança uma exceção (objeto) transferindo o controlo para o *handler* mais próximo na *stack* de execução capaz de apanhar exceções (objetos) do tipo do objeto lançado

- **`try`** `{ ... }`
  - executa bloco de instruções, sendo exceções apanhadas pelo(s) próximo(s) `catch`

- **`catch`** `(tipo-de-objeto [nome-de-objeto]) { ...}`
  - *handler*; apanha exceções do tipo indicado lançadas com `throw` no bloco de `try` ou em funções por ele chamadas

Tratamento de exceções manual *versus* automático

Manual

```
void f1()
{
    ...
    if (f2()==ERRO)
          goto TRATA_ERRO;
    ....
    return;

TRATA_ERRO:
    MsgBox("Erro .... ");
    ...
}
```

...

Automático

```
void f1()
{
    try
    {
        ...
        f2();
        ....
    }

     catch(ERRCOD e)
     {
         MsgBox("Erro ...");
         ...
     }
}
```

...

Tratamento de exceções manual *versus* automático (cont.)

Manual

```
...
```

```
ERRCOD f2()
{
    ...
    if (f3()==ERRO)
        return ERRO;
    ....
    return OK;
}
```

```
ERRCOD f3()
{
    ...
    if (f4()==ERRO)
        return ERRO;
    ....
    return OK;
}
```

```
...
```

Automático

```
...
```

```
void f2()
{
    ...
    f3();
    ....
}
```

```
void f3()
{
    ...
    f4();
    ....
}
```

```
...
```

Tratamento de exceções manual *versus* automático (conc.)

Manual

```
ERRCOD f4()
{
   ...
   if (situação de erro detectada)
        return ERRO;
   ....
   return OK;
}
```

Automático

```
void f4()
{
   ...
   if (situação de erro detectada)
        throw ERRO;
   ....
}
```

Conclusões:

- Com tratamento automático, só a função que deteta o erro (`f4`) e a função que pretende reagir ao erro (`f1`) é que têm código relacionado com o erro
- As funções intermédias na *stack* de chamada (`f2`, `f3`) não têm de fazer nada
- O lançamento (**`throw`**) do erro provoca o retorno automático (**`return`**) até à função que pretende reagir ao erro (intenção sinalizada com **`try`**) e um salto automático (**`goto`**) para o bloco de tratamento do erro (**`catch`**)

Exemplo com tratamento de exceções

```
class Data {
    int dia, mes, ano;
  public:
    class DiaInvalido { };   // define classe (tipo) de exceções
    void setDia(int dia);
    //...
};

void Data::setDia(int d) {
   if (d < 1 || d > 31) throw DiaInvalido(); // salta fora!
   dia = d;
}

main() {
    Data d;
    try {
       d.setDia(100);
        //....
    }
    catch (Data::DiaInvalido) {
      cout << "enganei-me no dia!!\n";
    }
}
```

lança objeto da classe *DiaInvalido* criado com chamada de construtor por omissão

handler

Agrupamento de exceções

- `catch ( ... )`
  - apanha uma exceção de qualquer tipo
- `catch ( T )`
  - apanha uma exceção do tipo `T` ou de um tipo `U` derivado publicamente de `T`
- `catch ( T * )`
  - apanha uma exceção do tipo `T *` ou do tipo `U *`, em que `U` é derivado publicamente de `T`
- `catch ( T & )`
  - apanha uma exceção do tipo `T &` ou do tipo `U &`, em que `U` é derivado publicamente de `T`

Exemplo com agrupamento de exceções

```
class Matherr { };
class Overflow  : public Matherr {};
class Underflow : public Matherr {};
class Zerodivide: public Matherr {};

void f()
{
    try {
       // operações matemáticas
    }
    catch (Overflow) {
       // trata exceções Overflow ou derivadas
    }
    catch (Matherr) {
       // trata exceções Matherr que não são Overflow
    }
    catch (...) {
      // trata todas as outras exceções (habitual no main)
    }
}
```

Exceções com argumentos; *re-throw*

```
class Data {
    int dia, mes, ano;
  public:
    class DiaInvalido {
     public:
         int dia;
         DiaInvalido(int d) {dia = d;}
    };
    void setDia(int dia);
    //...
};

void Data::setDia(int d) {
    if (d < 1 || d > 31)
            throw DiaInvalido(d);
    dia = d;
}
```

Exceções com argumentos; *re-throw*

```
void f() {
    Data d;
    try {
        d.setDia(100);
        // ...
    }
    catch (Data::DiaInvalido x) {
      cout << "dia inválido: " << x.dia << endl;
      throw;  // sem parêntesis => re-throw
    }
}
```

Tópicos adicionais

- Declaração de exceções lançadas por função:

  void data::setDia(int d) throw(DiaInvalido);

  void data::getDia(int d) throw(/*nenhuma*/);

  void data::setDate(int d, int m, int a) throw(DiaInvalido, MesInvalido, Ano Invalido, DataInvalida);

  (valor por omissão: pode lançar qualquer)

- Todo o corpo de uma função pode estar dentro de `try`:
  - `void f() try` { /* corpo*/ } `catch` ( ) { /**handler**/}

- Exceções não apanhadas fazem abortar o programa

- Exceções *standard* (hierarquia definida em <exception>)
  - bad_alloc, bad_cast, bad_typeid, bad_exception, out_of_range, invalid_argument, overflow_error, ios_base::failure