Tratamento de Excepções

João Pascoal Faria (versão original)
Ana Paula Rocha (versão 2004/2005)
Luís Paulo Reis (versão 2006/2007)

Introdução ao tratamento de excepções

- Excepções providenciam um modo de reagir a circunstâncias excepcionais (como erros de runtime) no nosso programa transferindo o controlo para funções especiais chamadas *handlers*.
- Pedaço de código é colocado debaixo da inspecção de excepções, num bloco (utilizando a palavra-chave "Try")
- Quando a circunstância excepcional aparece nesse bloco, uma excepção é lançada (palavra-chave "Throw") que transfere o controlo para o "handler" respectivo
- Se nenhuma excepção for lançada o código continua normalmente e todos os handlers são ignorados
- Os "handlers" são declarados com a palavra-chave "catch", imediatamente depois do bloco "Try".

Introdução ao tratamento de excepções

- Tratamento de excepções
  - maneira de transferir controlo e informação de um ponto na execução do programa, para uma rotina de tratamento da excepção (*handler*) associada a um ponto anterior na execução (retorno automático de vários níveis)
- **`throw`** `objecto;`
  - lança uma excepção (objecto) transferindo o controlo para o *handler* mais próximo na *stack* de execução capaz de apanhar excepções (objectos) do tipo do objecto lançado
- **`try`** `{ ... }`
  - executa bloco de instruções, sendo excepções apanhadas pelo(s) próximo(s) `catch`
- **`catch`** `(tipo-de-objecto [nome-de-objecto]) { ...}`
  - *handler*; apanha excepções do tipo indicado lançadas com `throw` no bloco de `try` ou em funções por ele chamadas
- A mudança de controlo obriga a desfazer a *stack* e a chamar os destrutores para os objectos automáticos construídos desde o `try`

Exemplos

```
// excepções exemplo
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;
    xxxxx;
    yyyy;
  }
  catch (int e)
  {
    cout << "Exception Nr. " << e << " occurred" << endl;
  }
  return 0;
}
```

```
Exception Nr. 20 occured
```

Exemplos

- Excepções int, char e default (nem int nem char) são apanhadas pelos handlers respectivos

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

- Possível encastrar múltiplos ciclos try (“nested exceptions”)

```
try { // code bloco 1 here
  try {
      // code bloco 2 here
  }
  catch (int n) { throw; }
}
catch (...) { cout << "Exception occurred";}
```


Tratamento de excepções 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 excepções 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 excepções 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 detecta 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`**)

Especificações de Excepções

- Ao declarar uma função podemos limitar o tipo da excepção que pode directa ou indirectamente lançar adicionando um sufixo à declarção da função:

```
float myfunction (char param) throw (int);
```

- Isto declara uma função designada myfunction que recebe um argumento do tipo char e retorna um elemento do tipo float. A única excepção que a função pode lançar é do tipo int.
- Se lançar uma excepção de tipo diferente não pode ser apanhada por um handler normal
- Se o especificador do tipo for deixado vazio sem tipo, a função não pode lançar excepções:

```
int myfunction (int param) throw(); // no exceptions allowed
int myfunction (int param); // all exceptions allowed
```


Exemplo com tratamento de excepções

```
class Data {
    int dia, mes, ano;
  public:
    class DiaInvalido { }; // define classe (tipo) de excepções
    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 objecto da classe `DiaInvalido` criado com chamada de construtor por omissão

handler

Agrupamento de excepções

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

Exemplo com agrupamento de excepções

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

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

Excepções 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;
}

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 excepções 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, AnoInvalido, DataInvalida);

  (valor por omissão: pode lançar qualquer)
- Todo o corpo de uma função pode estar dentro de `try`:
  - `void f() try { /* corpo*/ } catch ( ) { /*handler*/}`
- Excepções não apanhadas fazem abortar o programa
- Excepções *standard* (hierarquia definida em <exception>)
  - bad_alloc, bad_cast, bad_typeid, bad_exception, out_of_range, invalid_argument, overflow_error, ios_base::failure

Excepções Standard

- Biblioteca Standard C++ disponibiliza classe base para lidar com excepções #include <exception> (em namespace std)
- Classe possui os usuais construtores por defeito e cópia e destrutor
- Contém ainda um membro-função virtual designado "what" que retorna uma sequencia de caracteres (char*) que pode ser reescrita em classes derivadas e que contém a descrição da excepção

Exemplo – Excepções Standard

```
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e) { cout << e.what() << endl; }
  return 0;
}
```

```
My exception happened.
```

Excepções Standard

- Excepções lançadas por componentes da biblioteca std do C++ lançam excepções derivadas deste tipo de excepções (std::exception class) :

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

Excepções Standard

- Usando o operador new e não existindo memória suficiente para efectuar a alocação, excepção do tipo bad_alloc é lançada:

```
Try { int * myarray = new int[1000000000]; }
   catch (bad_alloc&) { cout << "Erro alocando memoria" << endl;}
```

- É recomendado colocar todas as alocações dinâmicas de memória dentro de um bloco try que apanha as excepções e termina o programa de modo limpo
- bad_alloc é derivada da classe base exception, por isso podemos apanhar referências à classe base:

```
class myexception: public exception {
  virtual const char* what() const throw()
    { return "My exception happened"; }
} myex;

int main () {
  try {  throw myex;
  }
  catch (exception& e) { cout << e.what() << endl; }
  return 0;
}
```

Objectos Funcionais

Objectos funcionais

- É util passar funções como argumentos

```
bool less_than_7(int v)
{ return v<7;
}

void function1(vector<int> & v)
{
  vector<int>::iterator it = find_if(v.begin(), v.end(), less_than_7);
  // …
}
```

- Mas muitas vezes é necessário que a função chamada guarde valores entre invocações sucessivas ⇒ **uso de classes**
- Objecto funcional
  - Objecto de uma classe com um operador de função

Objectos funcionais

```
template <class T>
class Sum
{
    T res;
  public:
    Sum (T i=0) : res(i) {}
    void operator() (T x) { res += x; }
    T result() const { return res; }
};


template <class T, class Op>
Op for_each(T first, T last, Op f)
{
    while (first != last)
      f(*first++)
    return f;
}
```


Objectos funcionais

```
int main()
{
    vector<int> v;
    v.push_back(2);
    v.push_back(5);
    v.push_back(8);
    v.push_back(3);
    Sum<double> s;
    s = for_each(v.begin(), v.end(), s);
    cout << "The sum is " <<s.result();  << endl;

}
```