Excepções em C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Motivação

Erros de execução a ter em conta na execução de programas:

- Input inválido (mal-formado, incompleto, ...)
- Erros de I/O (ficheiros, ligações de rede, dispositivos)
- Condições de execução anormais no ambiente (ex. falta de memória, espaço em disco)
- ...

Lidar com erros - estratégias possíveis

- Ignorar a possibilidade de erro.
- Uso de asserções.
- `abort` / `exit`
- Códigos de erro
- **Uso de excepções**

Lidar com erros ...

Ignorar a possibilidade de erro?

- No caso de erros ocorrerem, o programa não tem uma estratégia definida.
- Programa pode por exemplo terminar abruptamente de forma pouco graciosa ("crash") ou ter um fluxo de execução incorrecto.

Lidar com erros ... (cont.)

```
assert(this_should_never_happen());
```

Uso de asserções

O uso de `assert` é bastane útil durante desenvolvimento para validar que certas condições lógicas na execução de um programa *nunca* se verificam. O uso de asserções não é adequado para tratamento de erros *que podem* ocorrer.

Asserções não têm efeito se símbolo de pré-processamento `NDEBUG` ("no debug") estiver definido (ver documentação), como acontece frequentemente nas versões finais de software.

Tratamento de erros é de qualquer forma também simplista: programa pára imediatamente quando uma asserção não é verificada.

Lidar com erros ... (cont.)

```
if (error_condition) exit(1);
```

Uso de `exit` ou `abort`

Outra estratégia possível é programar uma chamada a `exit` ou `abort` no caso de um erro ser detectado.

O programa termina imediatamente. Esta estratégia é pouco adequada por ex. no contexto de código de bibliotecas que devem deixar ao programa cliente a melhor forma de lidar com erros.

Lidar com erros ... (cont.)

Uso de códigos de erro

```
int err = func();
if (err != 0) {
   . . . // trata erro
}
```

Possível erro é inferido pelo valor retornado por uma função.

Estratégia é empregue pelas funções da biblioteca de C ou funções POSIX do sistema operativo. Por exemplo, a função `printf` retorna um valor negativo no caso de erro de escrita ou de formato de impressão inválido.

Tratamento de erros tem também de ocorrer imediatamente à saída de uma função, o que pode levar a código verboso e a lógica de tratamento de erros muito dispersa.

Excepções

Em C++ podemos usar **excepções** para lidar com erros de forma estruturada. Outras linguagens como C# ou Java têm um suporte bastante similar.

- Erros são assinalados no ponto em que são detectados com o **lançamento de uma excepção** usando a instrução **`throw`**.
- **Excepção pode ser tratada** em outro ponto de código usando blocos **`try-catch`**.

Exemplo 1

```
try {
  std::cout << "Enter a positive integer: ";
  int n; std::cin >> n;
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  std::cout << "The number is " << n << std::endl;
}
catch(std::logic_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Execução normal

```
Enter a positive integer: 1
The number is 1.
```

Execução com excepção lançada

```
Enter a positive integer: 0
Error: expected positive integer
```

Exemplo 1 (cont.)

```
try {
  . . .
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  std::cout << "The number is " << n << std::endl;
}
catch(std::logic_error_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Excepção é lançada se `n <= 0`. Nesse caso, restantes instruções dentro do bloco `try` não são executadas e o fluxo de execução continua para o bloco `catch` que apanha a excepção, que é chamado o **"exception handler"**.

Excepção neste caso é um objecto do tipo `std::logic_error`, uma das classes de excepções definidas pela biblioteca C++ (a discutir mais à frente), e `what()` devolve mensagem de erro para a excepção.

Exemplo 2

```
int read_positive_int() {
  int n;
  std::cin >> n;
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  return n;
}
int main() {
   try {
     std::cout << "Enter a positive integer: ";
     int n = read_positive_int();
     std::cout << "The number is " << n << std::endl;
   }
   catch(std::logic_error& e) {
     std::cout << "Error: " << e.what() << std::endl;
   }
}
```

Exemplo 2 (cont.)

```
int read_positive_int() { . . .
  throw std::logic_error("expected positive integer");
. . . }
int main() {
   try {
     . . . int n = read_positive_int(); . . .
   }
   catch(std::logic_error& e) {
     std::cout << "Error: " << e.what() << std::endl;
   }
}
```

Execução é análoga ao exemplo 1, mas excepção é neste código lançada por `read_positive_int()` e tratada fora da função.

Quando lançada, a excepção leva a que `read_positive_int()` seja interrompida e que a execução continue no bloco `catch` de `main`. Restantes instruções dentro do bloco `try` em `main` não são executadas (`read_positive_int` não devolve um valor de retorno).

Exemplo 3

```
class time_of_day {
. . .
public:
  time_of_day(int h, int m) {
    if (h < 0 || h > 23 || m < 0 || m > 59)
      throw std::logic_error("invalid args");
    . . .
  }
```

Construtor de uma classe `time_of_day` para representar uma hora do dia valida se argumentos dados correspondem a horas e minutos válidos.

Excepção é lançada caso isso não aconteça. Objecto não chega a ser construído nesse caso.

Exemplo 3 (cont.)

```
try {
  time_of_day a(23, 59); // execução normal
  time_of_day b(24, 2); // lança excepção
  time_of_day c(23, 12); // não executado
} catch (std::logic_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

`b` e `c` não chegam a ser construídos.

Outro aspecto importante é que o destrutor de `a` é chamado antes do bloco `catch`. Para objectos definidos por uma função até ao lançamento da excepção, existe a garantia dos respectivos destrutores serem convenientemente invocados.

A instrução `throw` em sumário

Quando executamos

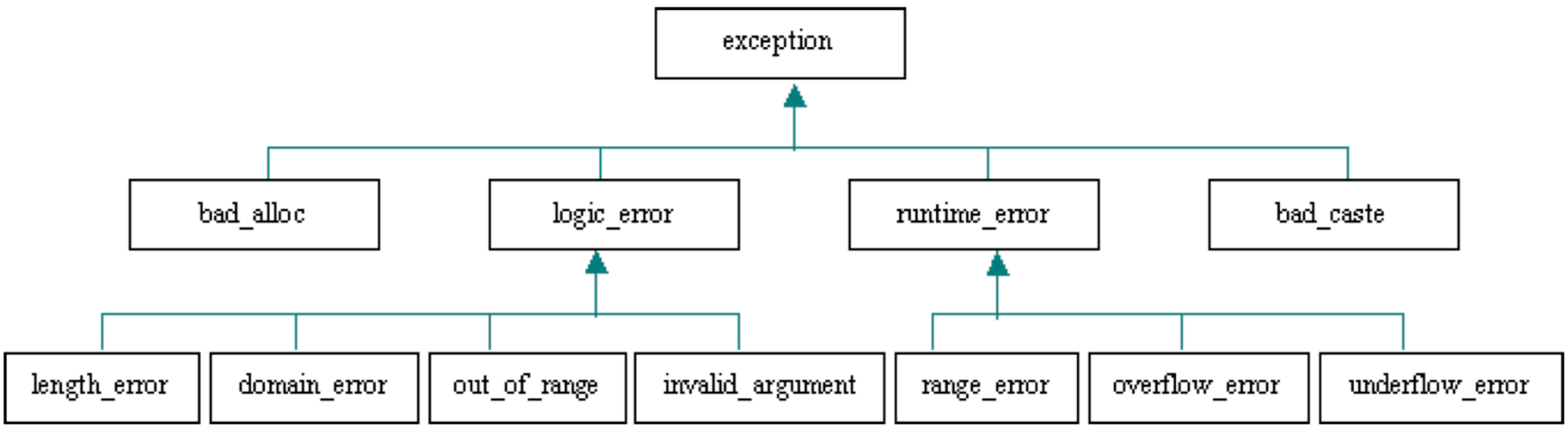
```
throw e;
```

- Fluxo de execução para a função actual é interrompido.
- Isto pode ocorrer sucessivamente para funções na "call stack". Execução retoma na primeira função na "call stack" que tenha um bloco `catch` para o tipo de `e`. "Call stack" é desfeita para cada função que seja necessária, com a garantia de que destrutores para objectos entretanto criados por cada função são devidamente chamados.
- No limite, se não houver um bloco `catch` adequado para processar a excepção, o programa termina por omissão imprimindo uma mensagem de erro (comportamento que é configurável, embora seja pouco usual fazer isso).

Tipos de excepções

```
throw e;
```

`e` pode ser de qualquer tipo (ex. até um valor `int`), mas normalmente é empregue uma classe de excepção definida no "header" `<stdexcept>` ou uma subclasse destas.



Tipos de excepções (cont.)

`logic_error`	exception class to indicate violations of logical preconditions or class invariants (class)
`invalid_argument`	exception class to report invalid arguments (class)
`domain_error`	exception class to report domain errors (class)
`length_error`	exception class to report attempts to exceed maximum allowed size (class)
`out_of_range`	exception class to report arguments outside of expected range (class)
`runtime_error`	exception class to indicate conditions only detectable at run time (class)
`range_error`	exception class to report range errors in internal computations (class)
`overflow_error`	exception class to report arithmetic overflows (class)
`underflow_error`	exception class to report arithmetic underflows (class)

`<stdexcept>`: define classes de excepção usadas pela biblioteca de C++. Por ex. `at` de `std::vector` pode lançar `std::out_of_range`:

Exceptions

`std::out_of_range` if `!(pos < size())`.

Tipos de excepções (cont.)

A função membro `what()` devolve uma string explicando a causa do erro:

```
try {
   . . .
}
catch(std::exception& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Excepções definidas pelo programador

Classes de excepções são tipicamente definidas como subclasses das classes standard de excepções.

Exemplo:

```
#include <stdexcept>
. . .
class invalid_time : public std::logic_error
{
public:
  invalid_time()
    : logic_error("invalid time") { }
};
```

Exemplo 4

Definição e uso de uma classe de excepção definida pelo programador:

```
class invalid_time : public std::logic_error {
public:
    invalid_time()
      : logic_error("invalid time") { }
};
class time_of_day {
. . .
public:
    time_of_day(int h, int m) {
        if (h < 0 || h > 23 || m < 0 || m > 59)
            throw invalid_time();
        . . .
    }
. . .
};
```

Múltiplos blocos `catch`

```
try {
  code_that_may_throw();
}
catch (type_1& e) {
  handle_type_1_exception(e);
}
catch (type_2& e) {
  handle_type_2_exception(e);
}
catch(...) {
  handle_other_exceptions();
}
```

Podemos ter vários blocos `catch` correspondentes a diferentes tipos de excepções. Um bloco `catch(...)` pode ser especificado por último para tratar excepções de um qualquer outro tipo.

Exemplo 5

Suponha que `f` pode lançar `std::runtime_error`, `std::logic_error` e `std::out_of_range`. Podemos definir individualmente blocos `catch` para cada uma dos tipos de excepção:

```
try {
  f(123);
}
catch(std::out_of_range& e) { . . . }
catch(std::logic_error& e) { . . . }
catch(std::runtime_error& e) { . . . }
```

Exemplo 5 (cont.)

Variantes (ver código dado): podemos agrupar o tratamento de `logic_error` e `out_of_range` num bloco para `logic_error` porque `out_of_range` é subclasse de `logic_error`

```
try {
  f(123);
}
catch(std::logic_error& e) { . . . }
catch(std::runtime_error& e) { . . . }
```

Exemplo 5 (cont.)

Variantes (ver código dado): podemos tratar todo os tipos de excepção no usando um único bloco `catch` para `std::exception`, já que é uma classe base de todas as outras no exemplo.

```
try {
  f(123);
}
catch(std::exception& e) { . . . }
```

ou então um único bloco `catch(...)` (que poderia apanhar ainda mais tipos de excepção caso fossem possíveis):

```
try {
  f(123);
}
catch(...) { . . . }
```