DCC024 Linguagens de Programação
2022.2

Tratamento de Erros

Haniel Barbosa

Tratamento de Erros

- Erros são comuns ao se escrever programas[*citation_needed*]
  1. Codificação errada
  2. Condições não tratadas durante codificação

Tratamento de Erros

- Erros são comuns ao se escrever programas[*citation_needed*]
  1. Codificação errada
  2. *Condições não tratadas durante codificação*

Tratamento de Erros

- Erros são comuns ao se escrever programas[*citation_needed*]
  1. Codificação errada
  2. *Condições não tratadas durante codificação*

- Como lidar com esses erros?

- Técnicas de tratamento de erros objetivam:
  - Prevenção de erros
  - Identificação de erros
  - Recuperação a partir de erros

Exemplo de tratamento de erros com pilhas

O que pode dar de errado com o *pop* de uma pilha? Como tratar o erro?

Exemplo de tratamento de erros com pilhas

O que pode dar de errado com o *pop* de uma pilha? Como tratar o erro?

```
1  class Stack                                          class Node
2  {                                                    {
3   private:                                             public:
4    Node* top;                                           unsigned data;
5                                                         Node* next;
6   public:                                              };
7    Stack() : top(nullptr) {}
8    unsigned pop()                          void push(unsigned n)
9    {                                       {
10     unsigned res = top->data;               Node* newTop = new Node();
11     Node* newTop = top->next;               newTop->data = n;
12     delete top; top = newTop;               newTop->next = top;
13     return res;                             top = newTop;
14   }                                       }
15   bool empty() { return top == nullptr; }
16   ~Stack()
17   {
18     while (top)
19     {
20       Node* next = top->next; delete top; top = next;
21     }
22   }
```

Exemplo de tratamento de erros com pilhas

```
1  int main()
2  {
3    Stack* stack = new Stack();
4    stack->push(5);
5    std::cout << "Popping... " << stack->pop() << "\n";
6  }
```

Exemplo de tratamento de erros com pilhas

```
1  int main()
2  {
3    Stack* stack = new Stack();
4    std::cout << "Popping... " << stack->pop() << "\n";
5  }
```

Introdução de uma guarda

```
1  int main()
2  {
3    Stack* stack = new Stack();
4    /* guard */
5    if (!stack->empty())
6    {
7      std::cout << "Popping... " << stack->pop() << "\n";
8    }
9  }
```

Totalizando a função

```
...
  unsigned pop()
  {
    /* total definition / error flagging */
    if (empty())
    {
      return -1;
    }
    ...
  }
  ...
};

int main()
{
  Stack* stack = new Stack();
  unsigned res = stack->pop();
  if (res != -1)
    std::cout << "Popping... " << res << "\n";
  else
    std::cout << "Can't pop an empty stack\n";
}
```

Falhas fatais

```
...
  unsigned pop()
  {
    /* fatal failure */
    if (empty())
    {
      std::cout << "FAILURE!!!\n";
      exit(-1);
    }
    ...
  }
  ...
};

int main()
{
  Stack* stack = new Stack();
  std::cout << "Popping... " << stack->pop() << "\n";
}
```

Pré condição

```
1  ...
2    unsigned pop()
3    {
4      /* pre condition */
5      assert(!empty());
6      ...
7    }
8    ...
9  };
10
11 int main()
12 {
13   Stack* stack = new Stack();
14   std::cout << "Popping... " << stack->pop() << "\n";
15 }
```

Exceções

```
...
  unsigned pop()
  {
    /* Exception */
    if (empty())
      throw 0;
    ...
  }
  ...
};

int main()
{
  try
  {
    Stack* stack = new Stack();
    std::cout << "Popping... " << stack->pop() << "\n";
  }
  catch (int i)
    std::cout << "Can't pop an empty stack\n";
}
```

Exceções

```
...
  unsigned pop()
  {
    /* Exception */
    if (empty())
      throw "Can't pop an empty stack\n";
    ...
  }
  ...
};

int main()
{
  try
  {
    Stack* stack = new Stack();
    std::cout << "Popping... " << stack->pop() << "\n";
  }
  catch (int i)
    std::cout << "Can't pop an empty stack\n";
  catch (const char* c)
    std::cout << c;
}
```

Exceções

```
1  class EmptyStackException : public std::exception
2  {
3   public:
4    std::string d_msg;
5    EmptyStackException(const std::string& msg) : d_msg(msg) {}
6    const char* what() const noexcept override { return d_msg.c_str(); }
7  };
```

Exceções

```
class EmptyStackException : public std::exception
{
 public:
  std::string d_msg;
  EmptyStackException(const std::string& msg) : d_msg(msg) {}
  const char* what() const noexcept override { return d_msg.c_str(); }
};
```

```
...
  unsigned pop()
  {
    if (empty())
      throw EmptyStackException("Can't pop an empty stack\n");
    ...
  }
...
int main()
{
  try ...
  catch(EmptyStackException e)
    std::cout << e.what();
}
```

Exceções em SML

```
1  fun fact n = if n = 0 then 1 else n * fact (n - 1);
2  fact 1;
3  fact 5;
4  fact ~1;
```

Exceções em SML

```
1  fun fact n = if n = 0 then 1 else n * fact (n - 1);
2  fact 1;
3  fact 5;
4  fact ~1;
```

```
1  exception BadArgument of int;
2
3  fun fact n =
4    if n < 0 then raise BadArgument n else if n = 0 then 1 else n * fact (n - 1);
5
6  fun useFact n =
7      "Answer = " ^ Int.toString (fact n)
8      handle BadArgument n => "Bad argument " ^ (Int.toString n);
9
10 useFact ~1;
```

Exceções em Python

```
1  class ArithmeticException(Exception):
2    def __init__(self, msg):          def __str__(self):
3      self.value = msg                  return repr(self.value)
4
5  def div(n, d):
6    if d == 0:
7      raise ArithmeticException("Attempt to divide " + str(n) + " by zero")
8    else:
9      return n/d
10
11 while True:
12   try:
13     n = float(input("Please, enter the dividend: "))
14     d = float(input("Please, enter the divisor: "))
15     r = div(n, d)
16     print("Result =", r)
17   except ValueError:
18     print("Invalid number format. Please, try again")
19   except ArithmeticException as ae:
20     print(ae.value)
21     break
```

Exceções em Python

```
1  class ArithmeticException(Exception):
2    def __init__(self, msg):          def __str__(self):
3      self.value = msg                  return repr(self.value)
4
5  def div(n, d):
6    if d == 0:
7      raise ArithmeticException("Attempt to divide " + str(n) + " by zero")
8    else:
9      return n/d
10
11 while True:
12   try:
13     n = float(input("Please, enter the dividend: "))
14     d = float(input("Please, enter the divisor: "))
15     r = div(n, d)
16     print("Result =", r)
17   except ValueError:
18     print("Invalid number format. Please, try again")
19   except ArithmeticException as ae:
20     print(ae.value)
21     break
22   finally:
23     print("We do this regardless")
```

Tipos de tratamento de erros

- Guardas
- Definições totais
- Sinalização de erros (*error flagging*)
- Finalização da execução (*fatal error*)
- Pré e pós condições
- Exceções (*exceptions*)

O uso de exceções define um padrão de projeto

- Declaração de exceções
- Geração de exceções
- Captura e tratamento de exceções

Exceções devem ser usadas com cuidado

- Introduzem um fluxo de controle implícito
  - Exceções são essencialmente *goto* sofisticados
  - Código com exceções pode ser mais difícil de ler e compreender
- Bons usos de exceções

Exceções devem ser usadas com cuidado

- Introduzem um fluxo de controle implícito
  - Exceções são essencialmente *goto* sofisticados
  - Código com exceções pode ser mais difícil de ler e compreender
- Bons usos de exceções
  - Repassar o erro para o método com o contexto necessário para tratá-lo

Exceções devem ser usadas com cuidado

- Introduzem um fluxo de controle implícito
  - Exceções são essencialmente *goto* sofisticados
  - Código com exceções pode ser mais difícil de ler e compreender
- Bons usos de exceções
  - Repassar o erro para o método com o contexto necessário para tratá-lo
  - Bloquear o fluxo de controle no ponto do erro é melhor do que continuar após ele

Exceções devem ser usadas com cuidado

- Introduzem um fluxo de controle implícito
  - Exceções são essencialmente *goto* sofisticados
  - Código com exceções pode ser mais difícil de ler e compreender
- Bons usos de exceções
  - Repassar o erro para o método com o contexto necessário para tratá-lo
  - Bloquear o fluxo de controle no ponto do erro é melhor do que continuar após ele
- Maus usos: todos os outros :)

Em resumo...

- Exceções são eventos "anormais" que podem ocorrer durante a execução de um programa, são caracterizados de maneira específica através de uma exceção, e permitem tratamento específico para tais situações.
- Mecanismos de tratamento de exceções em SML
  - Declaradas com *exception*, disparadas com *raise*, tratadas com *handle*
  - Possui exceções padrão (e.g., Div, Match, ...)
- Mecanismos de tratamento de exceções em Python
  - Blocos *try*/*except*
  - Extensões de *Exception*
  - Possui exceções padrão (e.g., ZeroDivisionError, ValueError, ...)
  - Bloco *finally* permite especificar ações padrão que sempre são executadas
- Mecanismos de tratamento de exceções em C++
  - Qualquer tipo. Mas recomendado usar extensões de *std::exception*
  - Blocos *try*/*catch*
  - RAII anula necessidade de *finally* para limpeza de memória