DCC024 Linguagens de Programação

2022.1

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?

```
class Stack
{
 private:
  Node* top;

 public:
  Stack() : top(nullptr) {}
  unsigned pop()
  {
    unsigned res = top->data;
    Node* newTop = top->next;
    delete top; top = newTop;
    return res;
  }
  bool empty() { return top == nullptr; }
  ~Stack()
  {
    while (top)
    {
      Node* next = top->next; delete top; top = next;
    }
  }
};
```

```
class Node
{
 public:
  unsigned data;
  Node* next;
};
```

```
void push(unsigned n)
{
  Node* newTop = new Node();
  newTop->data = n;
  newTop->next = top;
  top = newTop;
}
```

Exemplo de tratamento de erros com pilhas

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

Exemplo de tratamento de erros com pilhas

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

Introdução de uma guarda

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

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

```
...
  unsigned pop()
  {
    /* pre condition */
    assert(!empty());
    ...
  }
  ...
};

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

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

```
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(); }
};
```

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

```
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 ...
}
```

Exceções em SML

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

Exceções em SML

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

exception BadArgument of int;

fun fact n =
  if n < 0 then raise BadArgument n else if n = 0 then 1 else n * fact (n - 1);

fun useFact n =
    "Answer = " ^ Int.toString (fact n)
    handle BadArgument n => "Bad argument " ^ (Int.toString n);

useFact ~1;
```

Exceções em Python

```
class ArithmeticException(Exception):
  def __init__(self, msg):          def __str__(self):
    self.value = msg                  return repr(self.value)

def div(n, d):
  if d == 0:
    raise ArithmeticException("Attempt to divide " + str(n) + " by zero")
  else:
    return n/d

while True:
  try:
    n = float(input("Please, enter the dividend: "))
    d = float(input("Please, enter the divisor: "))
    r = div(n, d)
    print("Result =", r)
  except ValueError:
    print("Invalid number format. Please, try again")
  except ArithmeticException as ae:
    print(ae.value)
    break
```

Exceções em Python

```
class ArithmeticException(Exception):
  def __init__(self, msg):          def __str__(self):
    self.value = msg                  return repr(self.value)

def div(n, d):
  if d == 0:
    raise ArithmeticException("Attempt to divide " + str(n) + " by zero")
  else:
    return n/d

while True:
  try:
    n = float(input("Please, enter the dividend: "))
    d = float(input("Please, enter the divisor: "))
    r = div(n, d)
    print("Result =", r)
  except ValueError:
    print("Invalid number format. Please, try again")
  except ArithmeticException as ae:
    print(ae.value)
    break

  finally:
    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