UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação defensiva

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

- Direção defensiva

> Direção Defensiva é o ato de conduzir de modo a evitar acidentes, apesar das ações incorretas (erradas) dos outros e das condições adversas (contrárias), que encontramos nas vias de trânsito.
>
> – DETRAN

- Desenvolvimento de software
  - Queremos evitar acidentes (erros)
  - Apesar de ações incorretas (usuários)
  - Em condições adversas (resto do programa)

Introdução

- Programação defensiva
  - Não é ser defensivo sobre sua programação
  - "Eu garanto que funciona!"
- "Garbage in, garbage out"
  - Entradas ruins produzem saídas ruins
  - Colocar a culpa (obrigação) no usuário?
- Filosofia insuficiente (mal vista) atualmente
  - Marca de programas desleixados e não seguros
  - Alta disponibilidade ou segurança necessários

Programação defensiva

- Forma de design protetivo destinado a garantir o funcionamento contínuo de um software sob circunstâncias não previstas
  - "Garbage in, nothing out"
  - "Garbage in, error message out"
  - "No garbage allowed in"
- Erros mais fáceis de encontrar e corrigir e menos prejudiciais ao código de produção

Programação defensiva
Robustez vs. Corretude

- Robustez
  - Significa sempre tentar fazer algo que permita que o software continue operando, mesmo que isso às vezes leve a resultados imprecisos
- Corretude (exatidão)
  - Significa nunca retornar um resultado impreciso
  - Não retornar nenhum resultado será melhor do que retornar um resultado incorreto
- Qual característica deve ser priorizada?

Programação defensiva
Estratégias

- Validação das entradas
- Asserções
- Programação por contrato
- Barricadas
- Tratamento de exceções (próxima aula)

Validação das entradas

- Entradas
  - Sem controle, inesperadas e imprevisíveis
  - Podem ser inclusive mal-intencionadas
  - Assuma o pior de todas as entradas!
- Tratamento geral
  - Defina o conjunto de valores de entrada válidas
  - Ao receber entrada, valide com esse conjunto
  - Estabeleça um comportamento caso incorreta
    - Terminar / Repetir / Alertar

Validação das entradas
Exemplo 1

- O quão robusto é esse código?

```
int fatorial(int num) {
  int fat = 1;
  for (int i = 1; i <= num; i++)
    fat = fat * i;
  return fat;
}
```

- O que acontece nesse caso?

```
int main() {

  cout << fatorial(-2) << endl;

  return 0;
}
```


Validação das entradas
Exemplo 2

- Como esse se compara com o anterior?

```
int factorial(int num) {
  if(num == 0)
    return 1;
  else
    return num * factorial(num-1);
}
```

- Problema muito mais grave
  - Chamadas sucessivas até falta de recursos
  - Pode impactar outros sistemas
  - Como resolver?

Validação das entradas
Exemplo 3

- O quão robusto é esse código?

```
class Conta {

  int _agencia;
  int _numero;
  double _saldo;

  public:
    void depositar(double valor) {
      this->_saldo += valor;
    }
};
```

- Quais validações poderiam ser feitas?

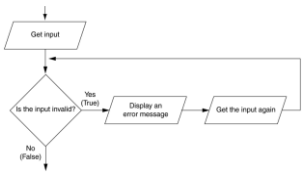
Validação das entradas
Exemplo 3

```
Conta c;
c.depositar('a');
c.depositar(-10);
c.depositar(9.99999);
c.depositar(1.0e-10);
```

- O valor é numérico?
- Aceita valores negativos?
- Número de casas decimais importa?
- Deve ser composto apenas de números?
- A quantidade é válida? (muito grande ou pequena)

Validação das entradas

- Validation loops → Error traps



- Comportamento depende da aplicação
  - Quantas vezes (e se) será executado?

Asserções

- Asserção/Assertiva
  - Predicado inserido para verificar (certificar) que determinada condição (suposição) é verdadeira
- Argumentos
  - Expressão booleana (deve ser verdadeira)
  - Mensagem a ser exibida caso não seja
- Códigos altamente robustos
  - Asserções (desenvolvimento) → Exceções (execução)

Asserções vs. Exceções

- Asserções
  - Erros fatais (não podem ser tratados)
  - Sempre indicam algum erro no código
  - Situações que nunca deveriam ocorrer
- Exceções
  - Situações excepcionais (podem ser tratadas)
  - Podem ocorrer mesmo em códigos corretos
    - Falta de memória, erro de comunicação, …
  - Na dúvida, faça uso de exceções

Asserções
Possíveis verificações

- Parâmetro de entrada está dentro do intervalo esperado
- Ponteiro a ser utilizado é não NULL
- Conteiner está vazio (ou preenchido) quando uma rotina começa a ser executada (ou quando termina)
- Conteiner usado em uma rotina deve/pode conter pelo menos (no máximo) um número X de elementos
- Arquivo ou stream está aberto (fechado) quando uma rotina começa a ser executada (termina a execução)

Asserções
Exemplo 1

```
//#define NDEBUG
#include <iostream>
#include <cassert>

using namespace std;

int main() {

  int vetor[10];
  for (int i = 0; i < 15; i++) {
    assert(0<=i && i<10);
    vetor[i] = i;
    cout << vetor [i] << endl;
  }

  return 0;
}
```

https://wandbox.org/permlink/LCx5hHq5ycY7hXpi

Asserções
Fail-fast programming

- Quando ocorrer um problema, o sistema deve falhar imediatamente e visivelmente
  - Evitar postergar uma falha para o futuro
- Ajudam a detectar erros precocemente
  - Análogas a fusíveis em um circuito
  - Falhar antes que mais danos sejam causados
  - Também ajudam a identificar a raiz de uma falha
- Asserções são desativadas no release (!)

https://martinfowler.com/ieeeSoftware/failFast.pdf

Programação por contrato

- Acordo entre duas ou mais partes
- Funções devem ser vistas como um contrato
  - Dada entrada, executam uma tarefa específica
  - Não devem fazer outra coisa além disso
  - Produzem alguma saída como resultado
- As funções podem receber qualquer entrada ou retornar qualquer saída? Quais impactos?
  - Como garantir a correta execução?

https://en.wikipedia.org/wiki/Design_by_contract

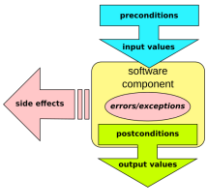
Programação por contrato

- Perguntas
  - O que o contrato espera?
  - O que o contrato garante?
  - O que o contrato mantém?
- Elementos (formalização lógica)
  - {Pré-condições} **ação** {Pós-condições}
  - {Invariantes}

Programação por contrato

- Pré-condições
  - O que deve ser verdadeiro para a rotina poder ser chamada (requisitos mínimos da rotina)
- Pós-condições
  - O que deve ser verdadeiro após a rotina executar
- Invariantes
  - Condições que devem sempre ser verdade, antes, durante e após a execução de uma região

Programação por contrato



- Utilize asserções para documentar e verificar pré-condições, pós-condições e invariantes

Programação por contrato
Exemplo 1

- O quão robusto é esse código?

```
class Conta {

  int _agencia;
  int _numero;
  double _saldo;

  public:
    void sacar(double valor) {
      this->_saldo -= valor;
    }
};
```

- Qual condição (mínima) deveria ser válida?
  - Gerar crash? → Exceção

Programação por contrato
Exemplo 1

- O código abaixo é suficiente?

```
if(conta.possuiSaldoSuficiente(valor)) {
  conta.sacar(valor)
}
```

- Não confiar nos usuários do método!

```
void sacar(double valor) {
  if (!possuiSaldoSuficiente(valor)) {
    throw ExcecaoSaldoInsuficiente();
  }

  this->_saldo -= valor;
}
```


Programação por contrato

- E se não for possível cumprir o contrato?
- Devem ter algum tipo de indicador
  - Definir um valor de erro global
  - Retornar um valor indicativo (inválido)
    - NULL?
    - False?
    - Número negativo?
  - Lançar uma exceção

Programação por contrato

- Pré-condições/Pós-condições vs. Herança?
- Subcontratação
  - A definição de uma subclasse significa uma extensão do contrato da superclasse
  - O contrato herdado pode ser redefinido desde que não viole o contrato da superclasse
- Pode-se enfraquecer as pré-condições e fortalecer as pós-condições de métodos
  - Condição mais fraca é menos restrita
  - Condição mais forte é mais restrita

Programação por contrato
Exemplo 2 – Pré-condições

```
class Conta {

  int _agencia;
  int _numero;
  double _saldo;

  public:

    virtual void depositar(double valor) {
      assert(valor > 0);
      this->_saldo -= valor;
    }
};
```

```
class ContaVIP : public Conta {

  public:

    void depositar(double valor) override {
      assert(valor > 50);
      this->_saldo -= valor;
    }
};
```


Programação por contrato
Exemplo 2 – Pré-condições

```
void doAction(Conta* conta) {
    conta->depositar(25);
}

int main() {

  Conta* c1 = new Conta();
  ContaVIP* c2 = new ContaVIP();

  doAction(c1);
  doAction(c2);

  return 0;
}
```

Programação por contrato
Exemplo 3 – Pós-condições

```
class Indexador {

  int _min = 5;
  int _max = 10;

  protected:
    virtual int getMin() {
      return _min;
    }

    virtual int getMax() {
      return _max;
    }

  public:
    virtual int getIndex() {
      return rand() % (getMax() - getMin())
             + getMin();
    }
};
```

```
class IndexadorEspecial : public Indexador {

  int _min = 0;
  int _max = 15;

  protected:
    int getMin() override {
      return _min;
    }

    int getMax() override {
      return _max;
    }
};
```

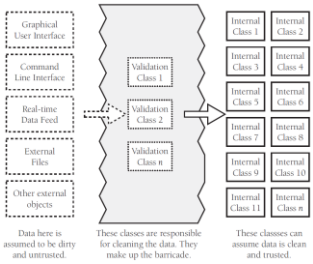

Programação por contrato
Exemplo 3 – Pós-condições

```
#include <ctime>

void doAction(Indexador* idx) {
    int v[] = {-1, -1, -1, -1, -1, 5, 6, 7, 8, 9};
    int id = idx->getIndex();
    cout << v[id] << endl;
}

int main() {
  srand((int)time(0));

  Indexador* idx1 = new Indexador();
  IndexadorEspecial* idx2 = new IndexadorEspecial();

  doAction(idx1);
  doAction(idx2);
}
```

Barricadas

- Crie barricadas no programa para minimizar o dano causado por dados incorretos
- Interfaces como limites para áreas "seguras"
  - Verifique todos os dados que cruzam os limites de áreas seguras, e informe se forem inválidos
- Partes que funcionam com dados *sujos* e algumas que funcionam com dados *limpos*
  - Responsabilidade pela verificação centralizada

Barricadas



Barricadas
Classes

- Métodos públicos
  - Validam os dados externos
- Métodos privados
  - Assumem que é seguro usar os dados
- Asserções vs. Exceções
  - Considere o uso de exceções para métodos públicos e asserções para métodos privados

Checklist

- ☐ Does the routine protect itself from bad input data?
- ☐ Have you used assertions to document assumptions, including preconditions and postconditions?
- ☐ Have assertions been used only to document conditions that should never occur?
- ☐ Does the architecture or high-level design specify a specific set of error handling techniques?
- ☐ Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- ☐ Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- ☐ Have debugging aids been used in the code?
- ☐ Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that's changed?
- ☐ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ☐ Is the amount of defensive programming code appropriate—neither too much nor too little?
- ☐ Have you used offensive programming techniques to make errors difficult to overlook during development?

Code Complete - Página 211

Considerações finais

- Regras gerais
  - Nunca assuma nada como verdade absoluta
    - Entradas, comportamento do usuário, recursos, …
  - Utilize padrões pré-estabelecidos
    - Codificação, design, documentação, …
  - Mantenha o código tão simples quanto possível
    - Deve conter apenas os recursos de que precisa
    - Complexidade é uma ótima fonte de erros
    - Planejamento adequado é essencial!

Considerações finais

- "Being Defensive About Defensive Programming"
- Críticas
  - Evitar uso excessivo de programação defensiva
  - Desperdício de tempo e dinheiro
    - Proteção de erros que nunca serão encontrados
  - Tente encontrar um equilíbrio
    - Aplicação → Robustez vs. Corretude

Considerações finais

- Quanta de programação defensiva deixar no código de produção?
  - Remova o código que resulta em falhas graves
  - Deixar código que verifica erros importantes
  - Mensagens de erro devem ser informativas
  - Registre (log) possíveis falhas (análise posterior)