Técnicas de Programação

Tratamento de Erros e Depuração de Código

Profa. Elaine Venson

elainevenson@unb.br

Semestre: 2012-1

Conteúdo

- Tratamento de erros
- Técnicas para teste de código
- Técnicas de depuração

Erros

- Erros **ocorrem**
- Quando não são tratados geram falhas/defeitos
- Sabe-se de antemão que podem ocorrer
- A **estabilidade** de um código depende de um tratamento de erros adequado
- Surgem em algum componente e são **comunicados** às camadas superiores para serem tratados por quem chamou o componente

Tipos de Erro

- Erro do usuário
  - Fornece valores de entrada inválidos
  - Realiza operação não prevista
- Erro do programador
  - Falha introduzida no código, “bug”
  - Idealmente nunca deve ocorrer
- Circunstâncias excepcionais
  - Eventos externos
  - Ocorrência de um evento inesperado
  - Falha de conexão, disco cheio, etc

Erros

- Para controlar a execução de um programa:
  - **Originar** um erro quando algo der errado
  - **Capturar** todas as possibilidades de erros
  - **Tratar** os erros adequadamente
  - **Propagar** erros que não podem ser tratados

Mecanismos de reporte de erro

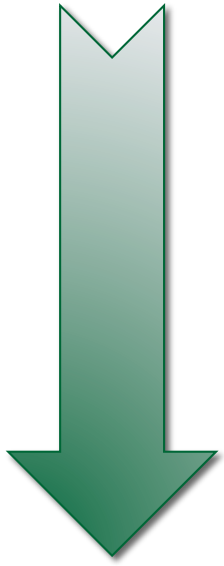
- Caráter local do erro:
  - *Local em tempo*: descoberto logo após ter sido criado
  - *Local em espaço*: identificado próximo ao local onde se manifestou

- Algumas abordagens visam reduzir a localidade do erro para identificá-lo mais facilmente (ex.: códigos de erro)

- Outras abordagem visam estender a localidade do erro para não misturar o código normal com lógica de tratamento de erros

Mecanismos de reporte de erros

- Em geral o mecanismo de tratamento de erros a ser aplicado é uma **decisão arquitetural**
- Mecanismos
  1. Não reportar
  2. Valores de retorno
  3. Variáveis de status de erro
  4. Exceções
  5. Sinais



Mais simples

Mais sofisticado

Mecanismos de reporte de erros

1) Não reportar:

- Não é uma solução viável
- Condições de erro não devem ser ignoradas
- Uma alternativa é abortar o programa instantaneamente, mas não é uma solução inteligente

2) Valores de retorno:

- Retornar valor indicando sucesso ou falha nas funções
- True ou false
- Enumeração de lista de códigos de erro, um código indicao sucesso e os demais os possíveis problemas

Mecanismos de reporte de erros

2) Valores de retorno (cont):

- Funciona bem para funções que não precisam retornar dados
- Para funções que retornam dados há algumas abordagens:
  - Retornar um tipo composto com o dados de retorno e o código de erro
  - Passar o código de erro através de parâmetro por referência
  - Reservar intervalo de valores para indicar erros (geralmente números negativos ou NULL para ponteiros)

Mecanismos de reporte de erros

3) Variáveis de status de erro:

- Utilizar variável de erro global compartilhada
- Após a chamada à função, a variável deve ser verificada para identificar possível erro ou sucesso
- Pode apresentar problemas de segurança inerentes às variáveis globais
- Por se tratar de uma variável separada da função é mais fácil de esquecer de atualizá-la ou verificá-la
- Biblioteca padrão do C possui a variável `errno`

Mecanismos de reporte de erros

4) Exceções:

- Recurso de algumas linguagens para tratamento de erros
- Quando é encontrado um problema que não pode ser tratado, a execução do código é interrompida e uma **exeção** é lançada
- A execução do programa vai retornando na pilha de chamadas até encontrar algum código de tratamento de exceção
- Exceção neste contexto é um **objeto** que representa o erro

Mecanismos de reporte de erros

4) Exceções

- Existem dois modelos de exceção de acordo com o que ocorre após o tratamento da exceção:
  - Modelo de conclusão: execução continua após o código que tratou a exceção
  - Modelo de recomeço: volta ao ponto onde a exceção foi lançada
- Uma exceção não pode ser ignorada, se não for tratada irá se propagar até o topo da pilha de execução e irá terminar a execução
- Exceções são em geral oferecidas por linguagens OO, os erros são definidos por uma hierarquia de exceções

Mecanismos de reporte de erros

5) Sinais:

- São **eventos** disparados pelo sistema operacional e enviados ao programa em execução na forma de sinais
- O sinal **interrompe** o fluxo normal de execução
- O programa pode receber um sinal a qualquer momento e deve ser capaz de tratá-lo
- Quando o tratamento do sinal é encerrado a **execução retorna ao** ponto onde havia sido interrompida
- Exemplo: um erro de ponto flutuante disparado pelo processador
- Deve ser instalado um handler para receber o sinal

Tratamento de erros

- Em que momento?
  - O mais rápido possível:
    - Melhor opção para funções que retornam códigos de erro
  - O mais tarde possível
    - Caso das exceções, permite fazer o tratamento para o erro no contexto mais adequado. Possibilidade de propagar o erro por vários níveis até ter as informações necessárias para tratar o erro

Tratamento de erros

- Ações possíveis:
  - Log do erro
    - Todo projeto grande deve ter um mecanismo de log
    - Existe para registrar eventos da execução da aplicação que podem ser de interesse para investigar problemas
  - Reportar
    - Os erros deve ser reportados ao usuário quando não há mais nada a fazer
    - Se é possível corrigir o erro, o usuário não precisa ser alertado

Tratamento de erros

- Ações possíveis (cont)
  - Reparar
    - Se não é possível tratar o erro em um determinado nível, a melhor alternativa pode ser passar para o nível acima
    - É provável que a função chamadora saiba como tratar o erro
  - Ignorar
    - Código que não inclui tratamento de erros tende a ter uma necessidade muito maior de depuração
  - Propagar
    - Duas alternativas: propagar a mesma informação ou reinterpretar
    - Reinterpretar de acordo com o contexto é uma técnica de código autoexplicativo

Exemplo – Versão 1

```
void nastyErrorHandling()
{
      if (operationOne())
      {
            ... do something …
            if (operationTwo())
            {
                  ... do something else ...
                  if (operationThree())
                  {
                        ... do more ...
                  }
             }
      }
}
```

Quanto mais operações, mais aninhado

Difícil de ler

Não reflete claramente as ações claramente (passos da operação x níveis)

Exemplo – Versão 2

```
void flattenedErrorHandling()
{
      bool ok = operationOne();
      if (ok)
      {
          ... do something ...
          ok = operationTwo();
      }
      if (ok)
      {
          ... do something else ...
          ok = operationThree();
      }
      if (ok) {
          ... do more ...
      }
```

Exemplo – Versão 2 (cont)

```
        if (!ok) {
        ... clean up after errors ...
    }
}
```

Não tem estruturas aninhadas

Adiciona variável

Adiciona código para rotina de limpeza

Exemplo – versão 3

```
void shortCircuitErrorHandling()
{
    if (!operationOne()) return;
    ... do something ...
    if (!operationTwo()) return;
    ... do something else ...
    if (!operationThree()) return;
    ... do more ...
}
```

Código mais simples e limpo

Não prevê código de limpeza

Uso de vários returns

Exemplo – Versão 4

```
void gotoHell()
{
    if (!operationOne()) goto error;
    ... do something ...
    if (!operationTwo()) goto error;
    ... do something else ...
    if (!operationThree()) goto error;
    ... do more ...
    return;
error:
... clean up after errors ...
}
```

Código mais simples e limpo

Prevê código de limpeza

Uso de `goto`

Exemplo – Versão 5

```
void exceptionalHandling()
{
     try
     {
          operationOne();
          ... do something ...
          operationTwo();
          ... do something else ...
          operationThree();
          ... do more ...
     }
     catch (...)
     {
          ... clean up after errors ...
     }
}
```

Pressupõe que as subfunções retornam exceções em vez de códigos de erro

Recursos são automaticamente desalocados

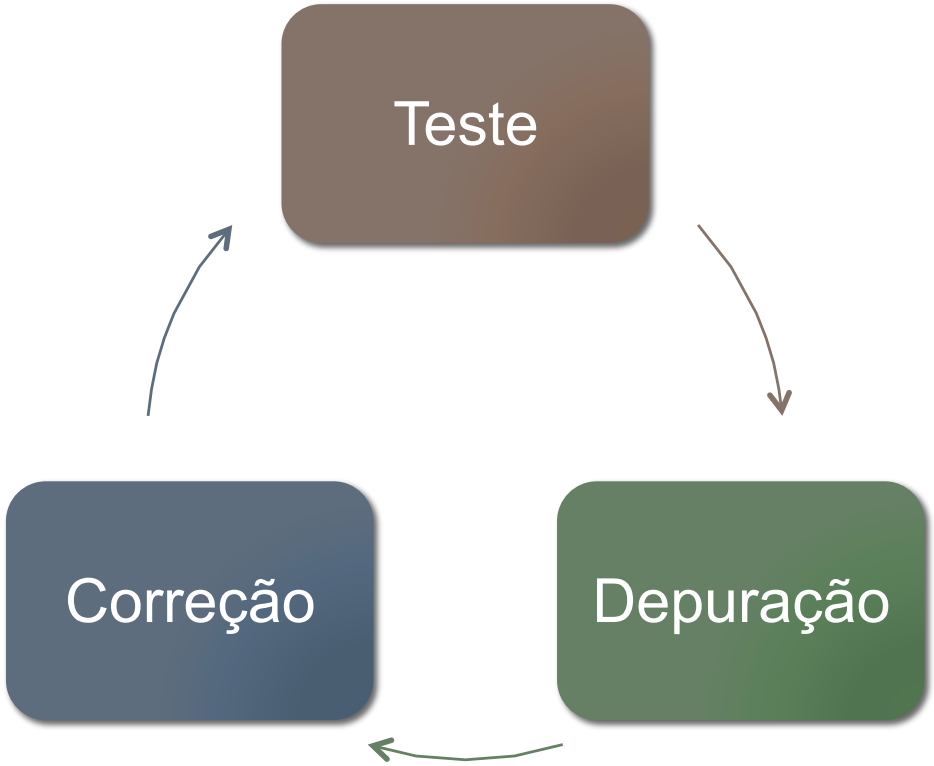
Checklist verificação erros

- Verificar todos os parâmetros da função
  - Considerar o uso de assertivas de acordo com o contexto
- Verificar se invariantes são satisfeitas em determinados pontos do programa
- Verificar a validade de todos os valores de fontes externas antes de utilizá-los
- Verificar o retorno de todas as chamadas a funções de sistema ou funções subordinadas


RATBERT, MY COMPANY IS HIRING FOR OUR QUALITY ASSURANCE GROUP. YOU'D BE PFRFECT.
WHAT WOULD I HAVE TO DO?
DILBERT®
BY
SCOTT ADAMS
YOU WOULD FIND FLAWS IN OUR NEW PRODUCT, THUS MAKING YOURSELF AN OBJECT OF INTENSE HATRED AND RIDICULE.
S.Adams E-mail: SCOTTADAMS@AOL.COM
7/1/96 © 1996 United Feature Syndicate, Inc.(NYC)
BUT THEN YOU'D FIX THOSE FLAWS... AND YOUR RESPECT FOR ME WOULD GROW INTO A SPECIAL BOND OF FRIENDSHIP, RIGHT?!
NO, THEN WE SHIP.

Processos Teste x Depuração

- **Teste**: provar a existência ou não de falhas no software
- **Depuração**: rastrear a causa da falha



Teste de código

- O processo de desenvolvimento de software prevê várias etapas de teste em vários níveis

- O foco desta aula é o **teste de código**, que ocorre ao longo de toda a atividade de construção do código

- Quem, o quê, quando e por que?

Teste de código

- Por que?
  - Encontrar as falhas e consertar
  - Garantir que as falhas não ocorram novamente
- Quem?
  - Responsabilidade do programador
  - Cada linha de código deve ser testada
- O que o teste involve? Considerar:
  - Definir exatamento que parte do código testar
  - Método de teste
  - Critérios de término

Teste de código

- Quando testar?
  - O teste deve ser realizado conforme o código é escrito
  - Neste momento os erros são mais fáceis de corrigir, afetam menos pessoas e garantem maior qualidade
  - Desenvolvimento orientado a teste
    - Métodos ágeis
    - Código de teste é escrito antes do código do software

Teste de código

- O teste de código deve demonstrar que:
  - Para todas as entradas válidas é gerada uma saída correta
  - Para todas as entradas inválidas é gerado o comportamento de erro adequado
- O conjunto de todas as entradas válidas e inválidas é geralmente muito grande
- Definir conjuntos representativos

Dificultadores do teste de código

- O conjunto de entradas válidas pode ser extremamente grande com apenas dois parâmetros
- Toda e qualquer estrutura de laço e condicionais adiciona complexidade
- Tamanho do código
- Dependências
- Entradas externas
- Estímulos externos

Dificultadores do teste de código

- Threads
- Evolução do código
- Problemas de hardware
- Modos de falha estranhos

- Solução:
  - Focar nos testes chave que conseguem capturar a maior parte dos defeitos

Tipos de teste

- Teste unitário
  - Foco em uma unidade atômica de código (classe ou função)
  - O código a ser testado é isolado – uso de stubs e simuladores
- Teste de componente
  - Combinação de unidades em um componente
- Teste de integração
  - Combinação de componentes assegurando que se integram corretamente

Tipos de teste

- Teste de regressão
  - Reteste realizado após correção de defeitos ou modificações no código ou no ambiente
- Teste de carga
  - Testar se o código comporta o volume de dados esperado para a aplicação em produção
- Teste de stress
  - Analisar o comportamento do código quando do envio de grande volume de dados em curto espaço de tempo
  - Utilizado para determinar a capacidade do software

Tipos de teste

- Teste de saturação
  - Semelhante ao teste de stress, porém conduzido em intervalo de tempo prolongado
  - Objetivo de identificar problemas de performance que ocorrem após execução de um grande número de operações
- Teste de usabilidade
  - Assegurar que o software pode ser utilizado facilmente pelos usuários finais

Abordagens para casos de teste

- Teste caixa preta:
  - Compara funcionalidade real com funcionalidade prevista
  - Funcionamento interno do código não é analisado
- Teste caixa branca:
  - “Teste estrutural”
  - Abordagem baseada na cobertura do código
  - Foco nas linhas de código em vez das especificações
  - Teste estático: não executa o código, apenas analisa
  - Teste dinâmico: buscar passar por todos os caminhos e ramificações possíveis

Abordagens para casos de teste

- Teste caixa branca (cont)
  - Muito mais trabalhoso e caro que o teste caixa preta
  - Utilização de ferramentas para instrumentar o teste e medir a cobertura do teste

- Para se obter um **teste unitário completo**, ambas as abordagens caixa preta e caixa branca são necessárias

Escolha dos casos de teste unitário

- Cada um dos casos de teste deve exercitar um aspecto diferente do código
- Necessário entender o requisito que foi implementado

- Casos de teste para Caixa Preta:
  - Entradas válidas:
    - testar fluxo normal
    - escolher valores médios e nos limites dos intervalos válidos

Escolha dos casos de teste unitário

- Casos de teste para Caixa Preta:
  - Entradas inválidas: incluir todos os tipos de entradas inválidas:
    - Números muito grandes ou muito pequenos
    - Dados de entrada muito extensos ou muito curtos
    - Valores inconsistentes
  - Valores limites:
    - Os valores limites
    - Valores imediatamente superiores
    - Valores imediatamente inferiores

Escolha dos casos de teste unitário

- Casos de teste Caixa Preta (cont)
  - Dados aleatórios
    - Aumentam as chances de encontrar erros inesperados
  - Zero
    - Por algum motivo, programadores falham em tratar zeros

Projetar para teste

- A forma como o código é **estruturado** pode facilitar ou dificultar o teste unitário
- Regras para facilitar:
  - Cada seção de código deve ser auto-contida
  - Evitar o uso de variáveis globais
  - Limitar a complexidade do código, decompondo-o em unidades menores
  - Fazer código “observável”

Automatização de teste

- Mais rápido e seguro
- Testes podem ser incorporados à rotina de gerar o buid, como uma etapa de validação
- Existem muitas ferramentes, como exemplo Junit (framework de teste unitário do Java)

Análise das falhas

- Quando o teste encontra uma falha, o problema precisa ser caracterizado antes de se iniciar a depuração:
  - O que estava sendo realizado no momento da falha e que eventos a dispararam
  - Verificar se o problema é repetível, com que frequência ocorre e se coincide com outras atividades realizadas no mesmo momento
  - Descrever a falha de forma específica (contexto, passos realizados, versão do build)
  - Registrar o defeito
  - Codificar um testador que demonstre a falha

Sistema de controle de defeitos

- Ferramenta com uma base de dados especializada em teste
- É atualizada conforme defeitos vão sendo identificados e posteriormente corrigidos
- Ações típicas:
  - Reportar defeitos
  - Atribuir responsável
  - Priorizar correção
  - Marcar defeitos corrigidos

Sistema de controle de defeitos

- Ações típicas (cont):
  - Encerrar defeito
  - Modificar defeito

- Existe grande número de ferramentas disponíveis comerciais e gratuitas, como o Bugzilla que é parte do projeto Mozilla