



## Aula 25 Instrumentação – Parte 1

Alessandro Garcia  
LES/DI/PUC-Rio  
Junho 2011

### Hoje... Especificação

- Objetivo dessa aula
  - Motivar instrumentação de programas
  - Introduzir o uso módulos de instrumentação
  - Como instrumentar para controle de espaços dinâmicos e cobertura de testes
- Referência básica:
  - Capítulo 14
  - Monografia

## Sumário



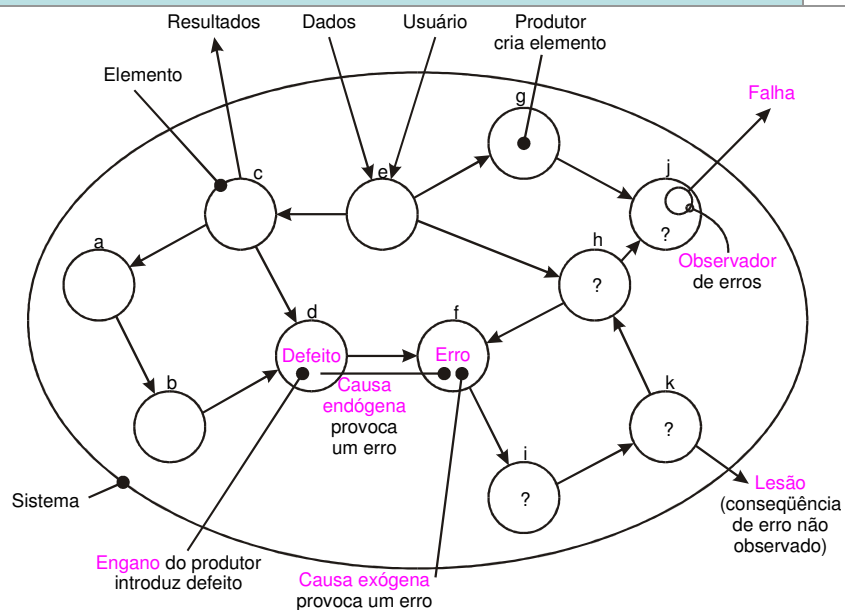
- Propriedades desejáveis de programas fidedignos
- Outros tipos de instrumentação
- Módulo de instrumentação (com adaptadores/*wrappers*)
  - Exemplo de módulo de instrumentação
- Necessidade do controle de espaços dinâmicos
  - características dos erros de uso de espaços dinâmicos
  - módulo de instrumentação CESP DIN
    - funcionalidade 1: simulação de memória insuficiente
    - funcionalidade 2: controle dinâmico de tipos

Jun 2009

Alessandro Garcia - LES/DI/PUC-Rio

3 / 41

## Qualidade do software em execução




Jun 2009

Alessandro Garcia - LES/DI/PUC-Rio

4 / 41

Laboratório de Engenharia de Software

## Qualidade do software em execução




- O **exercício do defeito** pode provocar um **erro de execução**
  - esse é um erro de **causa endógena** (**gerado internamente**)
  - devido a algum **engano do produtor** (humano, ferramenta) um elemento pode conter um **defeito** (falta, *fault*)
- O erro é **propagado** de elemento a elemento até que:
  - ou seja observado, neste momento passa a ser uma **falha** (*failure*)
  - ou o usuário seja afetado pelo erro, provocando uma **lesão**
    - o usuário pode **nem se dar conta** que ocorreu o dano
    - o usuário pode saber que ocorreu um dano, mas **não saber por quê** ocorreu
- Além de causas endógenas, erros podem ter **causas exógenas** (**gerado externamente**)
  - hardware, plataforma, outro software
  - mau uso

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
5 / 41

Laboratório de Engenharia de Software

## Qualidade do software em execução




Erros devidos a causas exógenas podem ser:

- **agressão** (condição hostil):
  - é um defeito / erro
    - ex.: vírus, cavalo de tróia, alteração do código, fraude (ex. *phishing*)
  - inserido / provocado **intencionalmente**
- **acidente** (condição adversa):
  - é um defeito inserido / erro provocado **sem intenção**
    - ex.: dado ou comando errado digitado acidentalmente pelo usuário
  - ou uma **falha causada por algum fator externo**
    - ex.: falta de energia, quebra de equipamento, etc...

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
6 / 41

Laboratório de Engenharia de Software

## Convicções – motivação para instrumentar



- **Não existe software sem defeitos**
  - o melhor que se consegue hoje (2009) é algo em torno de 0,8 defeitos para cada 10.000 linhas de código puro
    - não podem ser observados durante teste pois casos de teste nunca possuem 100% de cobertura do código
  - caso um software não contenha defeitos, não o saberemos
- **Erros exógenos** ocorrem com **probabilidade** (bem) **> zero**
  - 60 a 80% das falhas podem ser causadas por **erros de uso** humanos
- Pode custar cerca de 50% mais para desenvolver software de elevada fidelidade
  - mas este **investimento vale a pena** quando se considera o **custo total do software**
- **Práticas pessoais disciplinadas** podem reduzir a taxa de introdução de defeito em até 75 por cento
  - exemplo: instrumentação de código
  - Boehm, B.W.; Basili, V.R.; "Software Defect Reduction Top 10 List"; *IEEE Computer* 34(1); Los Alamitos, CA: IEEE Computer Society; 2001; pages 135-137


Jun 2009

Alessandro Garcia - LES/DI/PUC-Rio

7 / 41

Laboratório de Engenharia de Software

## Qualidade do software em execução



- Tanto lesões como falhas constituem **riscos de uso** do software
  - riscos têm uma **probabilidade** de ocorrer e geram um **dano**, exemplos de danos
    - retrabalho
    - perdas financeiras (ex. Pepsi China)
    - perda de material
      - peças mal fabricadas, produtos de baixa qualidade, decisão incorreta em virtude de informação não confiável
    - agressões à ecologia
    - perda de vida
    - perda de oportunidade (ex. sistemas de leilão eletrônico)
- Necessidade de INSTRUMENTAÇÃO para manter os riscos sob controle: **limite tolerável**

Jun 2009

Alessandro Garcia - LES/DI/PUC-Rio

8 / 41

## Caso 1 - USS Yorktown, 1997



- Cruiser USS Yorktown, 1997
  - faz o disparo e controle de mísseis guiados
  - erro exógeno por mal uso: membro da tripulação entrou um **valor ZERO**
  - **causa: nenhuma assertiva executável** no software que checasse tal violação
    - ou teste inadequado
  - o sistema de controle de software do navio:
    - **erros "em cascata"** devido a **divisão por zero**
      - vários módulos foram lesionados
    - causando **desligamento** do sistema de propulsão e saída por 2 horas
- mais informações sobre o acidente:  
<http://www.wired.com/science/discoveries/news/1998/07/13987>



## Caso 2 - Air France Flight 296




- Acidente envolvendo um Airbus 320
  - "Fly-by-wire" airbus
  - causas principais do acidente até hoje não são claras, mas relatório indica:
    - alguns **erros no software** manipulando dados do **Gravador de Dados Digitais** do vôo
    - **luz de emergência** não funcionou devido à um **erro de programação**
      - nenhuma **assertiva** que checasse que a altura para o solo do vô (10 metros) fugia do esperado (100 metros)
  - 3 passageiros morreram
- mais informações sobre o acidente:
  - [http://en.wikipedia.org/wiki/Air\\_France\\_Flight\\_296](http://en.wikipedia.org/wiki/Air_France_Flight_296)
  - <http://catless.ncl.ac.uk/Risks/9.63.html>
  - Youtube: <http://www.youtube.com/watch?v=2eQpUgHkBcg>



Laboratório de Engenharia de Software

## Tolerância a falhas: conceitos




- Para ser um software **fidedigno**, este deve ser robusto e ou tolerante a falhas
- Um programa **robusto** observa a ocorrência de erros endógenos ou exógenos
  - intercepta a execução quando observa um erro
  - mantém confinado o possível dano decorrente da falta
- Um programa **tolerante a falhas** (*fault tolerant*)
  - é um programa robusto
  - possui mecanismos de recuperação, habilitando-o a continuar operando confiavelmente (fidedignamente) após ter detectado uma falha
- **Deterioração controlada** (*graceful degradation*)
  - é a habilidade de um programa continuar operando corretamente após uma falha, embora com alguma perda de funcionalidade

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
11 / 41

Laboratório de Engenharia de Software

## Tolerância a falhas: conceitos




- Um programa **intolerante a falhas** **não** observa a ocorrência de erros endógenos ou exógenos
  - não é robusto
  - pode provocar lesões substanciais
  - **exemplo 1:** programas que não verificam a corretude dos dados e comandos fornecidos pelo usuário
  - **exemplo 2:** programas que contêm defeitos mas não contêm código de controle → faltam assertivas executáveis

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
12 / 41

Laboratório de Engenharia de Software

## Sistemas corretos fidedignos




- Um sistema correto é fidedigno se...
  - de que **posso justificavelmente depender** → em que posso confiar → digno de fé (fidedigno)
    - que **não provoca lesões** nem **falha de forma imprevisível**
- Logo, para ser um software correto fidedigno, tais **propriedades** são **desejáveis**...
  - capacidade de **detectar** o correspondente erro para evitar possíveis lesões
    - **detectabilidade**
  - a partir da informação relativa ao erro precisamos **diagnosticar** a falta que provocou o erro: descobrir as causas
    - **diagnosticabilidade**

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
13 / 41

Laboratório de Engenharia de Software

## Desafios para Propriedades desejáveis




- **Contribuem para dificultar a diagnose**
  - o **intervalo de tempo** decorrido desde o instante em que ocorreu o erro (exercício do defeito) até o instante em que o erro é observado
    - quanto maior esse intervalo mais custará determinar a causa
  - a **dificuldade de estabelecer com exatidão a causa** a partir dos problemas observados
    - algumas faltas não são visíveis diretamente “no código”
  - **características específicas** de domínios de aplicações ou bibliotecas
    - sistemas concorrentes e interfaces com “off-the-shelf components”
    - comportamento inesperado das bibliotecas, ex. condições de retorno não devidamente documentadas
  - **falhas intermitentes**
  - causas **externas ao código** que evidenciou a falha
    - extravasão de espaços (*array*, *buffer overflow*)
    - acidentes e agressões

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
14 / 41

Laboratório de Engenharia de Software

## Instrumentação: o que é



- A instrumentação
  - **monitora o comportamento** do programa em execução
    - *consome recursos de execução*
  - é formada por **fragmentos inseridos** no código
  - pode estabelecer e explorar **redundâncias**
    - exemplo: *N-Version Programming*
  - deve poder ser **inserida** e mais tarde **retirada** sem que isto afete a funcionalidade do programa
    - *não contribui para o serviço* a que se destina o programa
    - entretanto, ao *monitorar a correteude pode interceptar* a execução ao detectar uma falha
  - leva a custos adicionais...


Jun 2009

Alessandro Garcia - LES/DI/PUC-Rio

15 / 41

Laboratório de Engenharia de Software

## Instrumentação



- Objetivo - controlar a correteude **durante a execução**
  - **detectar** erros, isto é, identificar falhas:
    - o mais cedo possível
    - de forma automática, i.e. sem necessitar da intervenção humana
  - **impedir** que erros
    - propaguem lesões para outras partes do programa ou de outros sistemas
  - **simular e monitorar propriedades dinâmicas** do programa
    - simular mau funcionamento para fins de teste
      - *deturpadores*
    - cobertura dos testes
      - quanto do código foi exercitado durante os testes
    - vazamento de memória
    - ...

Jun 2009


Alessandro Garcia - LES/DI/PUC-Rio

16 / 41



Laboratório de Engenharia de Software

## Instrumentação: inserção no código




- A instrumentação **deve permanecer** no código
  - deve poder ser ativada ou desativada
    - **compilação condicional**
  - pode ser **útil ao alterar um programa** mais tarde
- Esquema de inclusão de instrumentos no código em C/C++
 

```
#ifdef _DEBUG
    • código do instrumento
    • funções e métodos de uso exclusivo em instrumentos
#endif
```
- Para que a instrumentação seja compilada o comando de compilação deve conter o parâmetro **/D\_DEBUG**
- É possível instrumentos com diferentes níveis de granularidade

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
17 / 41

Laboratório de Engenharia de Software

## Tipos de Instrumentos



- **Externos** ao programa
  - armaduras de teste ✓
  - depuradores
- **Internos** ao programa
 

assertivas executáveis  
 controladores de espaços de dados


  - contadores de passagem ← Aula de testes
  - verificadores de estruturas de dados ✓
  - deturpadores de estruturas de dados ✓

Jun 2009
Alessandro Garcia - LES/DI/PUC-Rio
18 / 41

Laboratório de Engenharia de Software

## Assertivas executáveis

- Assertivas executáveis contribuem para aumentar a **detectabilidade**
  - são capazes de observar a existência de um erro quase imediatamente após ter sido gerado
    - operam com uma frequência e rigor humanamente impossível
    - serão acionadas automaticamente *em cada caso de teste* executado
  - **transferem para a máquina** o controle da integridade dos dados e dos resultados
- Assertivas executáveis contribuem para aumentar a **diagnosticabilidade**
  - reduzem o esforço de diagnose




Nov 2009
LES/DI/PUC-Rio
19 / 41

Laboratório de Engenharia de Software

## Assertivas executáveis

- Assertivas executáveis contribuem para
  - aumentar a **detectabilidade**: identifica erro imediatamente
  - aumentar a **diagnosticabilidade**: reduz o esforço de diagnose
- É possível **traduzir** uma parcela considerável das assertivas **para código executável**. Exemplo:



$\forall pElem \in Lista \{ pLista \} : pElem \rightarrow pAnt \neq NULL \Rightarrow pElem \rightarrow pAnt \rightarrow pProx == pElem$

todos nós anteriores apontam para o próximo

AE: pLista aponta para a cabeça da lista a ser controlada  
 numErros = 0 ;  
 for ( pElem = pLista->pOrigem ; pElem != NULL ;  
     pElem = pElem->pProx )  
 {  
     if (pElem->pProx != NULL) {  
         if (pElem->pProx->pAnt != pElem)  
             numErros ++ ;  
     }  
 }  
 AS: se numErros != 0 a lista contém erros estruturais.

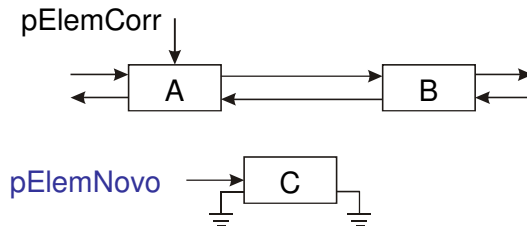
Teste é difícil

## Exemplo de assertivas executáveis

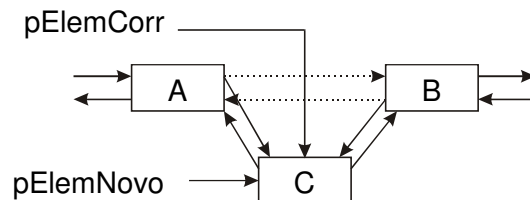


InserirApos( pElemCorr , pElemNovo )

Antes



Após



Nov 2009

LES/DI/PUC-Rio

21 / 41

## Exemplo de assertivas executáveis



```

void InserirElemApos( tpElem * pElemCorr , tpElem * pElemNovo )
{
    //Assertivas de entrada
    #ifdef _DEBUG
        tpElem * pElemAux ;
        tpElem * pElemCorrEntra ;
        if ( ( pElemCorr == NULL )
            || ( pElemNovo == NULL ) )
        {
            TratarErro( "InserirElem: Argumentos nulos" ) ;
        }
        if ( ( pElemNovo->pProx != NULL )
            || ( pElemNovo->pAnt != NULL ) )
        {
            TratarErro("InserirElem: Novo elem esta encadeado" ) ;
        }
        pElemCorrEntra = pElemCorr ;
        pElemAux = pElemCorr->pProx ;
        if ( pElemAux != NULL )
        {
            if ( pElemAux->pAnt != pElemCorr ) ← checa se o próximo elemento do elem
                                                corrente aponta para o nó corrente
            {
                TratarErro("InserirElem: Encadeamento da lista" ) ;
            }
        }
    }
    #endif
}
  
```

Nov 2009

LES/DI/PUC-Rio

22 / 41

## Exemplo de assertivas executáveis



```
// Efetuar a inserção

// Assertivas de saída
#ifdef _DEBUG
if ( !(( pElemNovo == pElemCorr )
    && ( pElemNovo->pAnt == pElemCorrEntra )
    && ( pElemNovo->pProx == pElemAux )
    && ( pElemCorrEntra->pProx == pElemNovo )))
{
    TratarErro( "InserirElem: Encadeamento antes" );
}
if ( pElemAux != NULL )
{
    if ( !( pElemAux->pAnt == pElemNovo ))
    {
        TratarErro( "InserirElem: Encadeamento apos" );
    }
}
#endif
} // Fim da função
```

checar se ponteiros  
do novo elemento  
estão corretos

checar se ponteiro  
do elemento à seguir  
aponta com pAnt para  
novo elemento

**Qual uma deficiência do uso de compilação condicional para realizar a instrumentação?**

atrapalha legibilidade do código/ algoritmo principal

Nov 2009

23 / 41

## Exemplo de assertivas executáveis



```
// Efetuar a inserção

// Assertiva de saída
#ifdef _DEBUG
if ( !(( pElemNovo == pElemCorr )
    && ( pElemNovo->pAnt == pElemCorrEntra )
    && ( pElemNovo->pProx == pElemAux )
    && ( pElemCorrEntra->pProx == pElemNovo )))
{
    TratarErro( "InserirElem: Encadeamento antes" );
}
if ( pElemAux != NULL )
{
    if ( !( pElemAux->pAnt == pElemNovo ))
    {
        TratarErro( "InserirElem: Encadeamento apos" );
    }
}
#endif
} // Fim da função
```

checar se ponteiros  
do novo elemento  
estão corretos

**Arcabouço fornece uma forma de contornar este problema parcialmente: AEs e ASS são implementadas como um *Wrapper* (Adaptador)**

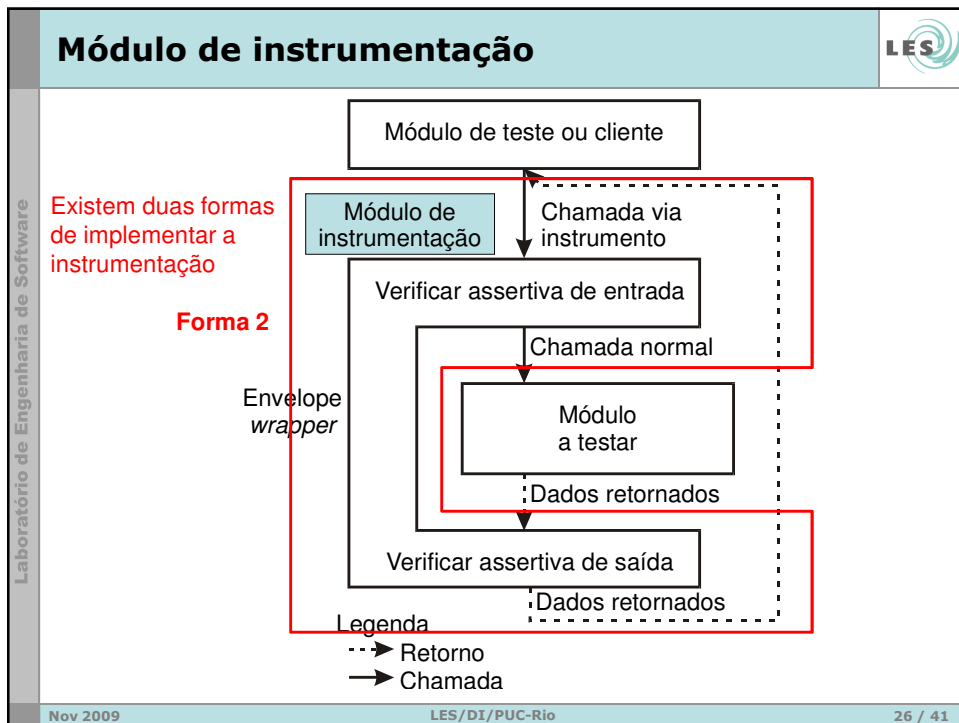
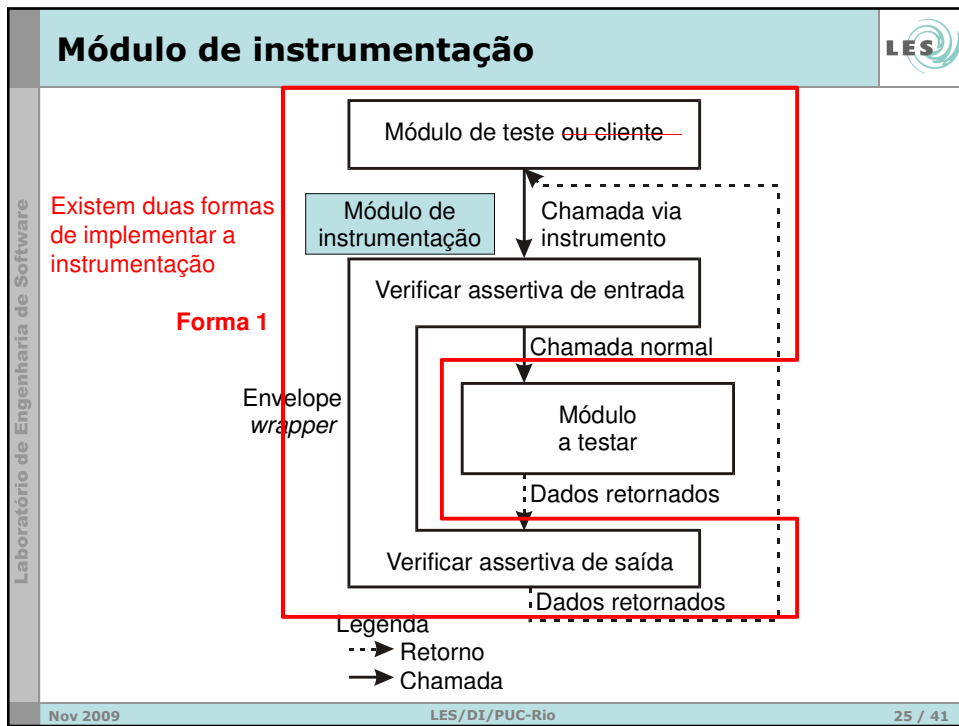
checar se ponteiro  
do elemento anterior  
aponta para novo  
elemento

**Qual uma deficiência desta forma de compilação condicional para realizar a instrumentação?**

atrapalha legibilidade do código/ algoritmo principal

Nov 2009

24 / 41



### FORMA 2:

**Passo 1.** Assume-se que módulo a testar e controlador de teste específico já foram desenvolvidos.

**Passo 2.** Deve ser criado um módulo de instrumentação

- este implementa as funções a monitorar com exatamente a mesma assinatura
  - sendo que o nome da função instrumentada passa a ter o sufixo **Instr**
- a implementação segue os padrões normais

**Passo 3.** O **módulo de definição** do módulo de **instrumentação** deve conter ao **final**:

```
#if !defined( modulo_instr_OWN )
#define Func1 Func1Instr
#define Func2 Func2Instr
. . .
#define FuncN FuncNInstr
#endif
```

Lista de todas as funções a serem controladas

**Passo 4.** O **módulo de teste** deve ser compilado com a chave **/D\_TESTE** (ou **/D\_DEBUG** ou outra qualquer)

- e deve conter o código:

```
#ifdef _TESTE
#include "modulo_instr.h"
#else
#include "modulo.h"
#endif
```

- O módulo a testar é implementado na forma convencional

## Módulo de instrumentação: exemplo



- teste\_main.c

```
#include <stdio.h>
```

```
#ifdef _TESTE
#include "teste_modulo_instr.h"
#else
#include "teste_modulo.h"
#endif
```

```
int main( int numParm , char ** vtParm )
{
    int numLidos = -1 ;
    ...

    printf( "\nSoma esperada: %d" , numSoma * ( numSoma + 1 ) / 2 ) ;
    res = Somar( numSoma , erro ) ;
    printf( "\nSoma obtida: %d\n" , res ) ;
}
```

fará a soma de um certo número (numSoma) de inteiros usando fórmula de Gauss

Usado para simular erro de execução

Ver o “exemplo de módulo de instrumentação” que se encontra no *site* da disciplina. **Aba Software**

## Módulo sendo testado: exemplo



- teste\_modulo.h

```
int Somar( int numSoma , int erro ) ;
```

Módulo de Definição (\*.h) do módulo a ser testado

- teste\_modulo.c

```
int Somar( int numSoma , int erro )
{
    return ( numSoma * ( numSoma + 1 ) / 2 + erro ) ;
}
```

Módulo de Implementação (\*.c) do módulo a ser testado

Laboratório de Engenharia de Software

## Módulo de instrumentação: exemplo

LES

- teste\_modulo\_instr.h

```

#if ! defined( teste_modulo_ )
#define teste_modulo_

#ifdef teste_modulo_instr_OWN
    #define teste_modulo_instr_EXT
#else
    #define teste_modulo_instr_EXT extern
#endif

int SomarInstr( int numSoma , int erro ) ;

#if !defined( teste_modulo_instr_OWN )
    #define Somar SomarInstr
#endif

#undef teste_modulo_instr_EXT

#endif

```

**Módulo de Definição (\*.h) da versão instrumentada do módulo**

```

#if !defined( teste_modulo_instr_OWN )
    #define Somar SomarInstr
#endif

```

←

Nov 2009

LES/DI/PUC-Rio

31 / 41

Laboratório de Engenharia de Software

## Módulo de instrumentação: exemplo

LES

- teste\_modulo\_instr.c

```

#include <stdio.h>
#include <assert.h>

#include "teste_modulo.h"

int SomarInstr( int numSoma , int erro )
{
    int res = -1 ;
    printf( "\nVerificar assertiva de entrada\n" ) ;
    assert( numSoma > 1 ) ;

    res = Somar( numSoma , erro ) ;

    printf( "\nVerificar assertiva de saída\n" ) ;
    assert( res == numSoma * ( numSoma + 1 ) / 2 ) ;

    return res ;
}

```

Deve preceder o módulo de definição instrumentado

←

AE

AS

Nov 2009

LES/DI/PUC-Rio

32 / 41



## Controle de espaços dinâmicos

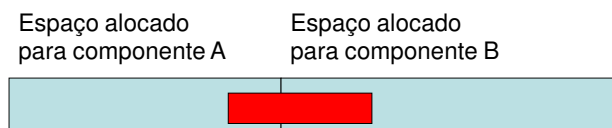


- Em linguagens que não possuem *garbage collection* pode ocorrer **vazamento de memória**
  - ocorre quando um espaço de memória deixa de ser acessível
    - *exemplo*: variável local; ao final da execução não é dado um 'free'
    - qualquer espaço não ativo é um espaço vazado
- Em linguagens que não possuem *garbage collection* pode ocorrer **acesso a espaços já desalocados**
  - podem existir vários ponteiros ativos para um mesmo espaço
  - um desses ponteiros pode ser usado para desalocar o espaço, enquanto os demais continuam apontado para área de memória real onde se encontrava o espaço desalocado
  - ao desalocar um espaço, o valor nele contido não é alterado
    - portanto, um programa com esse tipo de erro **pode continuar operando corretamente** !!! até que, muito tempo mais tarde, o espaço seja alocado para satisfazer uma nova requisição de alocação

## Controle do uso de espaços





- Várias linguagens permitem que se atribua valores a índices de elementos de um vetor além do final ou aquém do início do espaço declarado ou alocado
  - este tipo de erro pode interferir no funcionamento de outros componentes do programa
  - pode ocorrer de forma **intermitente**



Parcela do vetor de A que extravasou e invadiu B

Os espaços alocados são espalhados pela memória real e pode ocorrer que o espalhamento seja diferente de uma instância de uso para outra

Laboratório de Engenharia de Software	<b>Próxima aula</b>	
	<ul style="list-style-type: none"><li>• Uso do CESP DIN...</li></ul>	
Jun 2009	Alessandro Garcia - LES/DI/PUC-Rio	35 / 41

Laboratório de Engenharia de Software	<b>Trabalho 4</b>	
	<ul style="list-style-type: none"><li>• Enunciado: disponível até hoje a noite</li><li>• Ler seções 9 e 10 da Monografia do Arcabouço</li><li>• Estudar o módulo <b>CESP DIN</b> para controle de espaço dinâmico<ul style="list-style-type: none"><li>– entender as funcionalidades providas (*.h)</li></ul></li><li>• Estude o módulo <b>CONTA.h</b> do arcabouço</li><li>• Estude o exemplo contido na pasta “<i>Instrum</i>” do arcabouço<ul style="list-style-type: none"><li>– vide controlador de teste e scripts de teste de “Árvore”</li></ul></li><li>• Estudar outros exemplos (<b>disponíveis no sítio do curso – aba Software</b>)</li></ul>	
Nov 2009	LES/DI/PUC-Rio	36 / 41

Laboratório de Engenharia de Software

LES

FIM

Nov 2009

LES/DI/PUC-Rio

37 / 41