# Técnicas para desenvolvimento e aceleração de códigos científicos

**Raul Baldin**
LabMeC - FEC
UNICAMP

**Minicurso**
**LNCC 2014**

# Agenda

- Depuração de códigos
- Depuração usando GDB
- Detecção de Vazamento de Memória - Valgrind
- Laboratório (GDB+Valgrind)

# Depuração de códigos

# Depuração de códigos

**Técnicas mais utilizadas para depurar um programa:**

- Espalhar um monte de *printf* e *cout* pelo código
- Comentar partes do código suspeito de ter erro
- Comentar o código todo, ir “descomentando” linha a linha (ou binário) até que o programa quebre
- etc.

# Depuração de códigos

**Problemas:**

- Repetidas compilações p/ testes diferentes
- Não é possível pausar ou controlar a execução
- Somente mostra os valores das variáveis
- Não é possível trocar valores em tempo de execução
- etc

# Depuração de códigos

**Mais problemas:**

- Imagine fazer isso em programas grandes?
- Que usam inúmeras bibliotecas?

- Torna o processo de achar o erro chato e demorado!

e

- Nem sempre achamos o problema!

**Por que não utilizar uma ferramenta de depuração, como o GDB?**

# Agenda

- Depuração de códigos
- Depuração usando GDB
- Detecção de Vazamento de Memória - Valgrind
- Laboratório (GDB+Valgrind)

# Depuração usando GDB
# O que é?

- Ferramenta de depuração de códigos
- GDB → "*GNU Debugger*"
- Suporta outras linguagens além do C/C++, como Pascal, **Fortran**, etc.
- Permite pausar execução, continuar passo-a-passo, inspecionar, alterar variáveis
- Ajuda a encontrar os Bugs

# Depuração usando GDB
# Como usar?

- Compilar para depuração

```
g++ -g hello.cpp -o hello
```

  -g: adiciona símbolos de *debug*

- Carregar o programa no GDB:

```
gdb "./executável"
gdb "/caminho/do/executável"
```

# Depuração usando GDB
## O que são símbolos de depuração?

- Mapeiam instruções e seus:
  - Nomes de funções
  - Nomes / Tipos de variáveis
  - Números de linha no arquivo-fonte
  - Nome do arquivo-fonte

- **Vantagem**: facilita depuração
- **Desvantagem**: maior executável

# Depuração usando GDB
# Como usar?

```
raul@gatzalt:/tmp$ g++ -g hello.cpp -o hello
raul@gatzalt:/tmp$ gdb ./hello
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /tmp/hello...done.
(gdb)
(gdb)
```

# Depuração usando GDB
# Como usar?

- “(gdb)” é a linha de comandos
- Suporta auto-completar (TAB)
- Informações sobre os comandos:

```
(gdb) help comando
```

# Depuração usando GDB
# Como usar?

- Para carregar outro executável sem sair do GDB:

```
(gdb) file "outro_executável"
```

- Para executar o programa carregado:

```
(gdb) run (argumentos)
```

# Depuração usando GDB
# Como usar?

```
raul@gatzalt:/tmp$ g++ -g hello.cpp -o hello
raul@gatzalt:/tmp$ gdb ./hello
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /tmp/hello...done.
(gdb) run
Starting program: /tmp/hello
Hello, o conteudo de a = 1
[Inferior 1 (process 22055) exited normally]
(gdb)
```

# Depuração usando GDB
# Exibindo código-fonte

- Mostra o código do programa, posicionado na <linha>:

```
(gdb) list <linha>
```

- Mostra o código do programa, posicionado na <função>:

```
(gdb) list <nome_funcao>
```

# Depuração usando GDB
# Exibindo código-fonte

```
(gdb) list 0
1       #include <iostream>
2
3       int main () {
4
5       int a = 1;
6
7       std::cout << "Hello, o conteudo de a = " << a << std::endl;
8
9
10      }
(gdb) list main
1       #include <iostream>
2
3       int main () {
4
5       int a = 1;
6
7       std::cout << "Hello, o conteudo de a = " << a << std::endl;
8
9
10      }
(gdb)
```

# Depuração usando GDB
# Breakpoints

- Marcadores posicionados onde queremos pausar e inspecionar o código
- Para criá-los:

```
(gdb) break <num_linha>
(gdb) break <nome_arq>:<num_linha>
(gdb) break <nome_funcao>
```

# Depuração usando GDB
# Breakpoints

- Exemplos:

```
(gdb) break 10
(gdb) break programa.c:10
(gdb) break main
```

- Breakpoints condicionais:

```
(gdb) break 10 if i > 10
```

# Depuração usando GDB
# Breakpoints

- Para listar os breakpoints criados:

```
(gdb) info breakpoint
```

- Para apagar breakpoints:

```
(gdb) delete <#num_breakpoint>
```

- Para habilitar/desabilitar breakpoints:

```
(gdb) disable <#num_breakpoint>
(gdb) enable <#num_breakpoint>
```

# Depuração usando GDB
# Controlando a execução

- Continua execução até que encontre outro breakpoint ou termine

```
(gdb) continue
```

- Continua a execução, roda a próxima linha de código e pausa

```
Obs: Se prox linha = função, entre nela e execute-a linha-a-linha
```

```
(gdb) step
```

# Depuração usando GDB
# Controlando a execução

- Continua a execução, roda a próxima linha de código e pausa

```
Obs: Se prox linha = função, trate-a como única linha
```

```
(gdb) next
```

- Continua execução até termino do laço

```
(gdb) until
```

- Continua a execução até função atual terminar

```
(gdb) finish
```

# Depuração usando GDB
# Watchpoints

- Similar ao breakpoint, pausa a execução se valor de variável mudar

```
(gdb) watch variável
(gdb) info watchpoints
(gdb) delete #watchpnt_id
```

# Depuração usando GDB
# Exibindo informações

- Mostra valor da variável uma única vez:

```
(gdb) print variável
```

- Mostra valor da variável uma única vez (em hexa):

```
(gdb) print/x variável
```

- Mostra valor da variável em cada passo de execução:

```
(gdb) display variável
```

- Lista os "displays" configurados e seus #ids:

```
(gdb) info display
```

- Apaga um “display”:

```
(gdb) undisplay #id_display
```

# Depuração usando GDB
# Exibindo informações

- Atribui valor a variável (do escopo):

```
(gdb) set variable nome_var = 1
```

- Executa funções do programa ou bibliotecas, dentro do GDB:

```
(gdb) call nome_funcao()
```

- Desvia o fluxo de execução:

```
(gdb) jump num_linha
(gdb) jump endereco_mem
```

# Depuração usando GDB
# Exibindo informações

- Lista a pilha de chamadas (*Call Stack*) com seus #frame_nums:

```
(gdb) backtrace
(gdb) backtrace NUM ou -NUM
```

- Troca o quadro (*frame*) - permite inspecionar quadro:

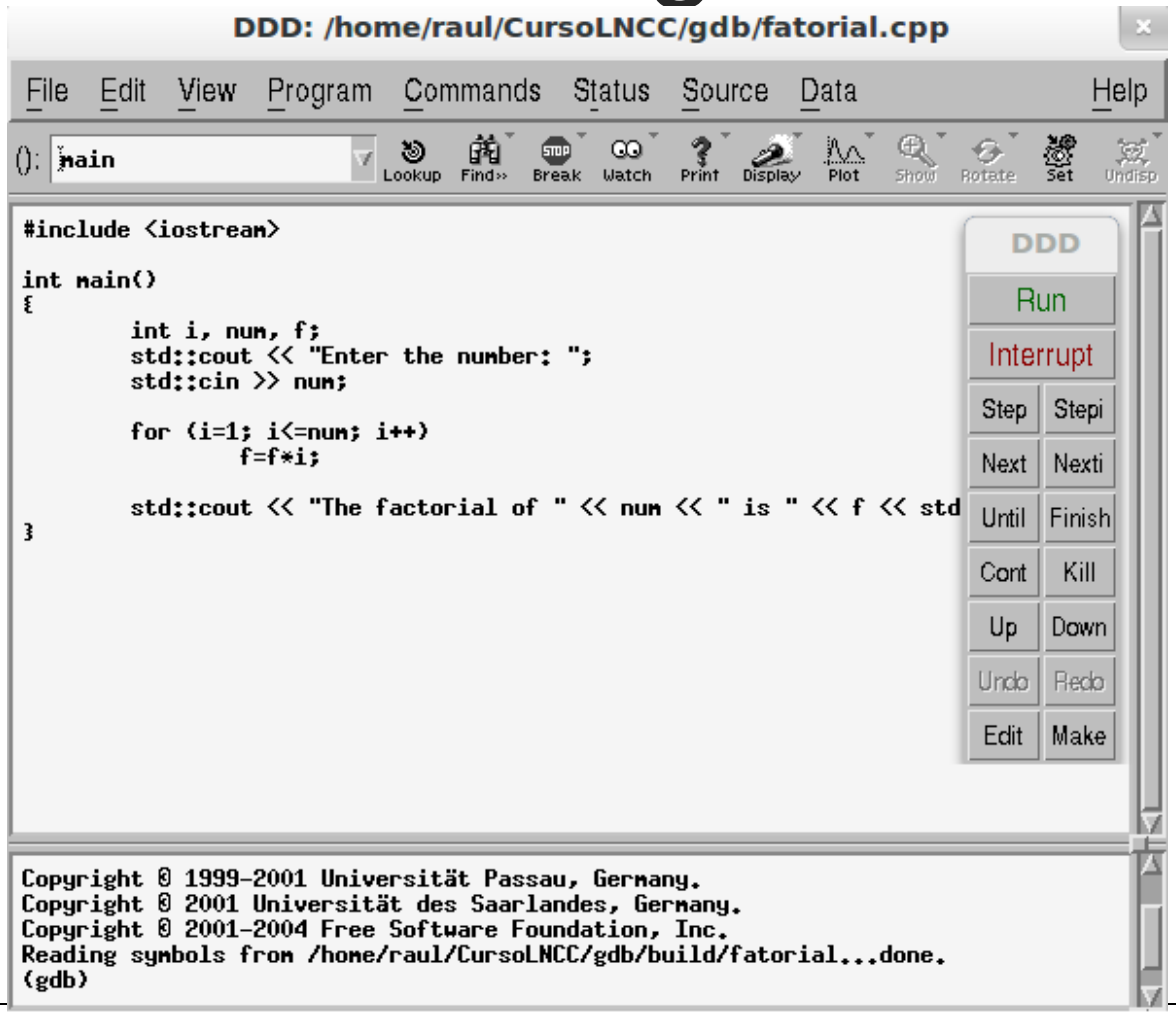
```
(gdb) frame #num_frame
```

- Faz o quadro selecionado retornar (com valor ou não) para quem o chamou:

```
(gdb) return valor_opcional
```

→ `Simulando fim/retorno de uma função, por exemplo.`

# Depuração usando GDB
# Interface gráfica



# Agenda

- Depuração de códigos
- Depuração usando GDB
- Detecção de Vazamento de Memória - Valgrind
- Laboratório (GDB+Valgrind)

# Vazamentos de memória

- **O que é um vazamento de memória?**
- É uma perda de um espaço de memória
- Espaço é alocado (*malloc* ou *new)* e nunca desalocado

- Se isso acontece várias vezes um código longo ou recursivo, causa problemas aleatórios e difíceis de encontrar (dependem do fluxo de entradas e execução).

**Por que não usar o Valgrind para detectar esses problemas?**

# Valgrind

- **O que é a ferramenta Valgrind?**
- Valgrind é um detector de falhas no gerenciamento de memória
- Mostra vazamentos de memória, erros alocação/desalocação, etc
- É um conjunto de ferramentas (***memcheck****, cache profiling*, etc)

# Valgrind – Como usar?

- Compilar com as opções:

```
gcc -g -Wall -o programa programa.c
```

```
-g : inclui símbolos de debug
-Wall : mostra todos os warnings
```

- Executar o programa pelo Valgrind:

```
valgrind --leak-check=yes ./exemplo
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Uso de memória não inicializada (Lab ex01)**

```c
#include <stdio.h>

int main()
{
    //Declarando a variavel
    int semInicializacao;
    //Tentando imprimir o valor da variavel declarada, porem nao inicializada: Comportamento indefinido!
    printf ("%i\n", semInicializacao);

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Acesso à memória depois de desalocá-la (Lab ex02)**

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

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *usarEndDesalocado;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    usarEndDesalocado = new int();
    //Inicializando memoria com o valor 1111
    *usarEndDesalocado = 1111;
    //Desalocando memoria
    delete usarEndDesalocado;
    //Tentando imprimir o valor de memoria previamente desalocada: Comportamento indefinido!
    printf ("%i\n", *usarEndDesalocado);

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Acesso à memória fora do espaço alocado (Lab ex03)**

```
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *usarEndForaAlocacao;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    usarEndForaAlocacao = new int();
    //Inicializando memoria com o valor 1111
    *usarEndForaAlocacao = 1111;
    //Tentando imprimir o valor de memoria alem do espaco alocado: Comportamento indefinido!
    printf ("%i\n", *(usarEndForaAlocacao+1));

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Acesso à memória fora do espaço alocado - vetor (Lab ex03.1)**

```c
#include <stdio.h>

int main()
{
    //Declarando um vetor de 5 posicoes 'int'
    int intArray[5] = {1,2,3,4,5};
    //Tentando acessar a 6a. posicao: Comportamento indefinido!
    printf ("%i\n", intArray[6]);

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Uso errado de malloc/new/new [] vs free/delete/delete [] - 2 chamadas de delete sobre um mesmo ponteiro (Lab ex04)**

```
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *duplodelete;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    //Utilizando o operador NEW
    duplodelete = new int();
    //Inicializando memoria com o valor 1111
    *duplodelete = 1111;
    //Liberando endereco de memoria
    delete duplodelete;
    //Liberando o mesmo endereco de memoria: PROBLEMA!
    delete duplodelete;

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Uso errado de malloc/new/new [] vs free/delete/delete [] - usando free() em endereço alocado por new (Lab ex04.1)**

```
#include <stdlib.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *int_usando_new;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    //Utilizando NEW
    int_usando_new = new int();
    //Inicializando memoria com o valor 1111
    *int_usando_new = 1111;
    //Liberando endereco de memoria
    //Utilizando FREE: Nao utilizacao do par certo para desalocar memoria
    //New -> Delete
    //Malloc -> Free
    free (int_usando_new);

    return 0;
}
```

# Valgrind

- O que o ***memcheck*** pode detectar/mostrar?
- **Vazamentos de memória: onde os ponteiros p/ endereços de memória previamente alocados foram perdidos (Lab ex05)**

```
#include <stdio.h>

int main()
{
    //Declarando um ponteiro para tipo 'int'
    int *valor_inteiro_ptr;
    //Alocando memoria de tamanho 'int' e gravando seu endereco no ponteiro
    valor_inteiro_ptr = new int();
    //Inicializando memoria com o valor 1111
    *valor_inteiro_ptr = 1111;
    //Nova alocacao de memoria, endereco para a antiga alocacao sera perdido!
    //Perdemos o endereco anteriormente alocado
    valor_inteiro_ptr = new int();
    //Desalocando endereco mais recentemente alocacao
    //Endereco alocado anteriormente nao foi desalocado: MEMORY LEAK!
    delete valor_inteiro_ptr;

    return 0;
}
```

# Valgrind - Interface Gráfica

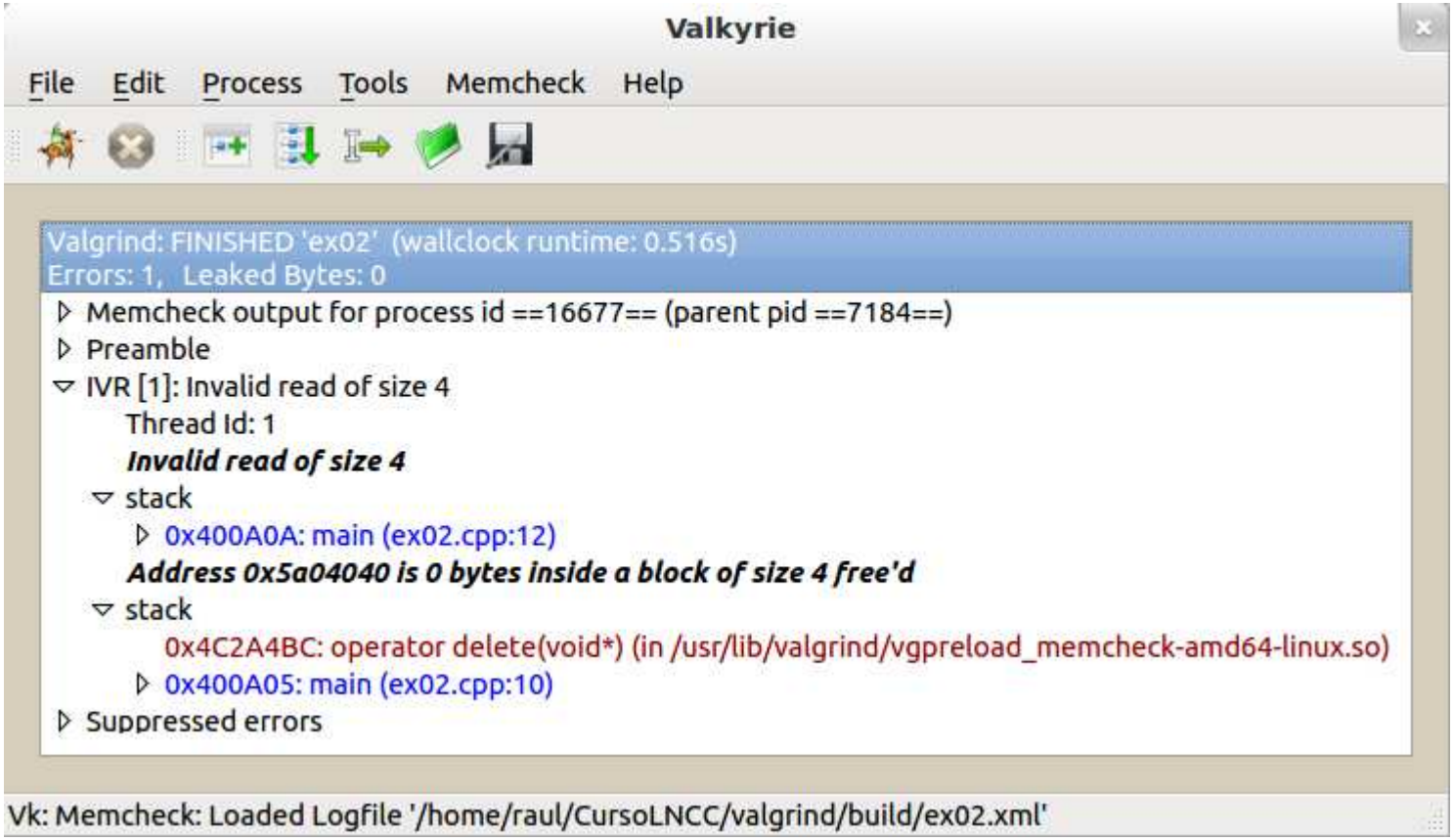
- Exportando resultados do Valgrind:

```
valgrind --leak-check=yes --xml=yes
--xml-file=prog.xml ./exemplo
```

- Interface para análisar resultados:

```
valkyrie -l  prog.xml
```

# Valgrind - interface Valkyrie



# Agenda

- Depuração de códigos
- Depuração usando GDB
- Detecção de Vazamento de Memória - Valgrind
- Laboratório (GDB+Valgrind)

# Atividades de Laboratório

**GDB:**

http://www.ic.unicamp.br/~edson/disciplinas/lncc14/lab-gdb.html

**Valgrind:**

http://www.ic.unicamp.br/~edson/disciplinas/lncc14/lab-valgrind.html

# Links recomendados

https://www.gnu.org/software/gdb/gdb.html

https://sourceware.org/gdb/onlinedocs/gdb/

https://www.gnu.org/software/ddd/

http://valgrind.org/

http://valgrind.org/docs/manual/quick-start.html

http://valgrind.org/downloads/guis.html

Valgrind+Fortran:
http://people.sc.fsu.edu/~jburkardt%20/f_src/valgrind/valgrind.html