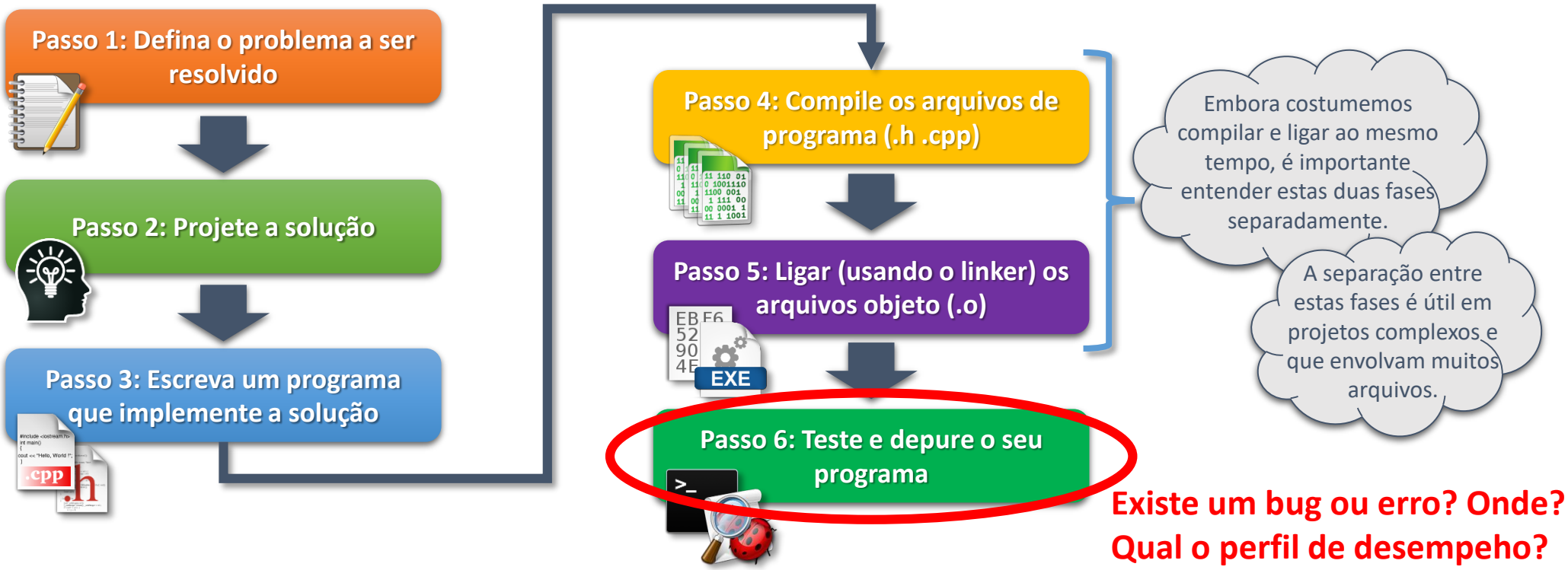
IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 03 – Depuração (com o GDB) e Profiling (com o Gprof)

Introdução

- Relembrando…



Depuração de Código

- Encontrar erros em programas pode se tornar uma tarefa difícil e exigir muito tempo
- Todo programador deveria procurar
  - Escrever código de boa qualidade
  - Estudar e aplicar técnicas que ajudam a evitar erros
  - Estar ciente das características das linguagens utilizadas
  - Escrever bons casos de testes
  - **Usar boas ferramentas de depuração**
- Depuração (debugging ou debug) é um procedimento para diagnóstico e correção de **erros já detectados** em um programa
- Em geral, mais da metade do tempo gasto no desenvolvimento de software é gasto com depuração

Depuração de Código

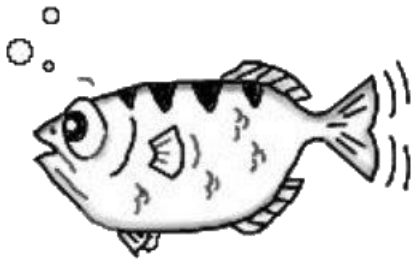
- A depuração é muito útil pois permite ao programador
  - Monitorar a execução de um programa (passo a passo)
  - Ativar pontos de parada (linha, função, condição)
  - Monitorar os valores das variáveis
  - Alterar áreas de memória
  - Monitorar as chamadas de funções (*stack trace*)
  - Retroceder na execução do programa
- Apesar da boa utilidade, a depuração não é sempre a melhor alternativa
  - Certos tipos de programas não se dão muito bem com ela
  - Sistemas operacionais, sistemas distribuídos, múltiplos threads, sistemas de tempo real
  - Não é possível em algumas linguagens e ambientes
  - Pode variar muito de um ambiente para outro
  - Pode ser complicada para programadores iniciantes

Depuração de Código

- Uma solução alternativa à depuração se passa pelo uso criterioso dos comandos de impressão (printf, cout)
  - Estrutura de mensagens que mostra por onde o código está passando
  - No GNU gcc/g++ podemos simplificar o trabalho de depuração incluindo a macro __FUNCTION__ que imprime o nome da função que esta sendo executada
    - Exemplo (gcc): printf ("Entrou na função : %s\n", __FUNCTION__);
    - Exemplo (g++): cout << "Entrou na função : " << __FUNCTION__ << endl;
  - Mas o **código original é alterado** para poder inserir estas mensagens 
- Um grande interesse da depuração é que ela permite mostrar por onde o código está passando **sem alterar o código original** 

Depurador GNU Debugger (GDB)

- GDB é um depurador do GNU que suporta diversas linguagens de programação
  - C, C++, Objective-C, Java, Fortran, etc.
- GDB foi criado por **Richard Stallman** em 1986
- Ele é mantido pelo comitê GDB nomeado pela Free Software Foundation
- Tal qual os compiladores GCC e G++ do GNU, o depurador GDB também pode ser integrado à diversos ambientes de desenvolvimento
  - NetBeans, Eclipse, Code::Blocks, Dev-C++, etc.



Usando o GDB em linha de comando

- As diretivas do compilador e do ligador (linker) permitem:
  - Inserir informações de depuração no código do programa (**-g**)
  - Indicar ao compilador que toda forma de otimização deve ser eliminada a fim de facilitar o funcionamento do depurador (**-O0**)
- Assim, para depurar um programa em C++ usando o **gdb** deve-se:
  - Compilar o programa com as diretivas de compilação **-g** e **-O0**
    - Exemplo: # **g++ -O0 -g -o programa main.cpp programa.cpp**
  - Carregar o programa no ambiente do **gdb**
    - Exemplo: # **gdb programa**
  - Uma vez no prompt do **gdb**, utiliza-se os comandos do gdb para a depuração
    - **(gdb)**

Comandos do GDB

- Os comandos podem ser indicados por sua abreviação, normalmente dado por uma letra
- Comandos de propósito geral
  - **help | h** : ativa a ajuda do gdb
  - **run | r**: executa o programa do início
  - **quit | q**: sai do gdb
  - **kill | k** : interrompe a execução do programa
  - **list linha | l linha** : lista partes do código fonte
  - **show listsize & set listsize N** : mostra & configura a qtde de linhas mostradas no comando list

Comandos do GDB

- Comandos de controle de execução
  - **break linha | b linha** : adiciona um ponto de parada na linha especificada do arquivo principal
  - **break funcaoX | b funcaoX** : adiciona um ponto de parada no inicio da funcaoX do arquivo corrente
  - **break arq.cpp:linha | b arq.cpp:linha** : adiciona um ponto de parada na linha especificada do arquivo fonte de nome **arq.cpp**
  - **info break | i b** : lista informações sobre os pontos de parada definidos
  - **delete N | d N** : remove o ponto de parada N (visualize o valor de N com o comando **info break**) – o comando **delete** sem a indicação do ponto de parada permite remover todos os pontos de parada de uma só vez
  - **disable N** : não remove, mas desativa o ponto de parada N
  - **enable N** : reativa o ponto de parada N

Comandos do GDB

- Comandos de controle de execução
  - **continue | c** : continua a execução do programa até o próximo ponto de parada
  - **continue N | c N** : continua, mas ignora o ponto de parada atual N vezes
  - **finish**: continua até o final da função
  - **step | s** : executa a próxima instrução (entrando na função) - **step N** executa as próximas N instruções
  - **next | n** : executa a próxima instrução (não entra na função) - **next N** executa as próximas N instruções
  - **backtrace | bt** : mostra onde estamos na pilha de execução
  - **backtrace full | bt f** : imprime os valores das variáveis locais
- Comandos de visualização da pilha de execução
  - **frame** : mostra o quadro (frame) corrente na pilha de execução
  - **up** : movimenta um quadro para cima
  - **down** : movimenta um quadro para baixo

Comandos do GDB

- Comandos de impressão:
  - **print VAR | p VAR** : imprime o valor armazenado na variável VAR
  - **print/x VAR | p/x VAR** : imprime o valor armazenado na variável VAR em formato hex
    - É possível usar outros formatos: **/d - int**; **/o - oct**; **/u - uint**; **/b - bin**; **/c - char**; **/f - float**
  - **ptype VAR** : imprime o tipo da variável VAR
- Aqui, são apresentados os comandos mais comuns. Há uma lista extensa de outros comandos ou variações que podem ser usados. O GDB ainda permite depurar no nível de linguagem de máquina, permitindo inspecionar posições de memória e código de máquina.
  - A documentação completa do GDB encontra-se disponível em:
    - **http://www.gnu.org/software/gdb/documentation/**
  - Um guia de referência rápida para o GDB encontra-se disponível em:
    - **https://web.stanford.edu/class/cs107/gdb_refcard.pdf**

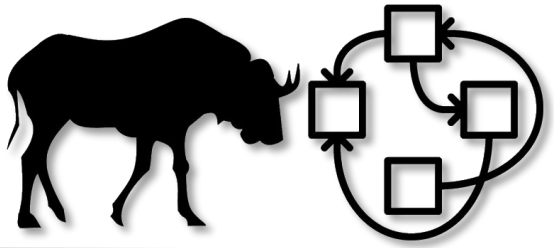
GDB na prática

- **Usando o GDB, aponte os problemas e as devidas correções para o código a seguir.**

```
# include <iostream>

int main()
{
        int i, num, j;
        std::cout << "Entre com um valor inteiro: ";
        std::cin >> num;

        for (i=1; i<num; ++i)
                j=j*i;

        std::cout << "O fatorial de " << num << " eh: " << j << std::endl;

        return 0;
}
```

Profiling

- Um **profiler** é um programa utilizado para avaliar o desempenho do seu programa, permitindo encontrar os gargalos (pontos onde o programa demora mais)
  - Apresenta, entre outras informações, um gráfico com o tempo de execução de cada função
  - Essa ferramenta também permite conhecer as funções presentes no código, o número de chamadas dentro de cada função e a porcentagem de tempo gasto em cada uma delas
- O **gprof** (GNU Profiling) é uma ferramenta que faz parte do GCC (GNU Compiler Collection), desenvolvida por Jay Fenlason, que serve para medir o tempo gastos pelas funções de um algoritmo, e exibi-las
  - Originalmente escrito por um grupo liderado por Susan L. Graham na University of California, Berkeley para o Berkeley Unix (4.2BSD)

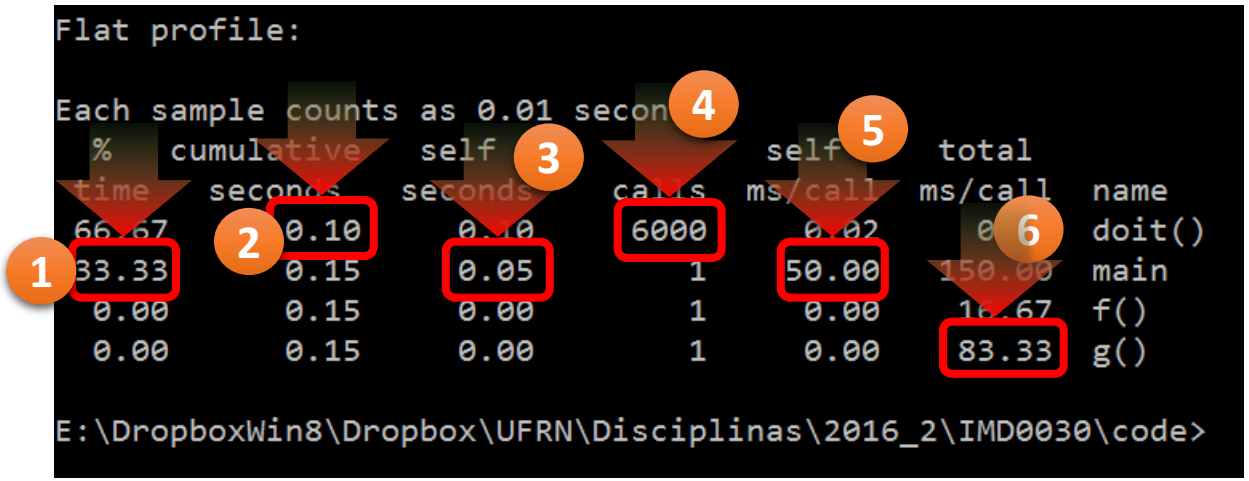


Gprof

- Para usar o **gprof**
  - Primeiro deve-se compilar o programa incluindo a diretiva de compliação **-pg**, que fará com que o compilador insira informações adicionais para o profiler
  - Executar o programa uma vez para que seja criado o profile **gmon.out** que será lido pelo **gprof**
  - Executar o **gprof** sobre o executável
    - Exemplo 1: **$ gprof --brief -p programa**
      - Neste exemplo as informações de profiling serão mostradas na tela
      - Muitas vezes é muita informação na tela (difícil de ler!)
    - Exemplo 2: **$ gprof –brief -p programa > profile.log**
      - Neste exemplo as informações de profiling serão armazenadas no arquivo profile.log

Gprof

- **Flat profile**: **gprof --brief -p <programa>**
  - Exibe informações sobre a quantidade total de tempo gasto pelo programa na execução de cada função
  - Exemplo: **$ gprof --brief -p ./prog**



1. Percentagem do tempo total gasto na execução da função principal (main).
2. Tempo gasto executando esta função e as demais acima desta.
3. Quantidade de segundos contados apenas para esta função.
4. Quantidade de vezes que a função foi invocada.
5. Quantidade média de *ms* gasta por chamada a esta função sozinha.
6. Quantidade média de *ms* gasta por chamada a esta função e suas predecessoras.

Gprof

- **Call graph**: **gprof --brief -q <programa>**
  - Exibe informações sobre as chamadas das funções ao longo da execução do programa
    - Permite acompanhar a ordem de execução do programa
  - Exemplo: **$ gprof --brief -q ./prog**
  - Resultados:
    - A função g() foi chamada pela função main() apenas 1/1 vez
    - A função g(), por sua vez, chamou a função doit() 5000/6000 vezes

```
                     Call graph


granularity: each sample hit covers 4 byte(s) for 6.67% of 0.15 seconds

index % time    self  children    called     name
                0.05    0.10       1/1           _GLOBAL__sub_I__Z4doitv [2]
[1]    100.0    0.05    0.10       1         main [1]
                0.00    0.08       1/1           g() [4]
                0.00    0.02       1/1           f() [5]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00    0.15                 _GLOBAL__sub_I__Z4doitv [2]
                0.05    0.10       1/1           main [1]
-----------------------------------------------
                0.02    0.00    1000/6000        f() [5]
                0.08    0.00    5000/6000        g() [4]
[3]     66.7    0.10    0.00    6000          doit() [3]
-----------------------------------------------
                0.00    0.08       1/1           main [1]
[4]     55.6    0.00    0.08       1          g() [4]
                0.08    0.00    5000/6000        doit() [3]
-----------------------------------------------
                0.00    0.02       1/1           main [1]
[5]     11.1    0.00    0.02       1          f() [5]
                0.02    0.00    1000/6000        doit() [3]
-----------------------------------------------

Index by function name

   [5] f()                     [3] doit()
   [4] g()                     [1] main
```

Gprof

- Maiores detalhes sobre o uso do **gprof** podem ser encontradas em:
  - https://sourceware.org/binutils/docs/gprof/
- Exemplos de uso do **gprof** podem ser encontradas em:
  - http://www.thegeekstuff.com/2012/08/gprof-tutorial/

Gprof na prática

- Programa exemplo:
  - **$ g++ -Wall -pedantic -std=c++11 -g -O0 -pg -o prog prog.cpp**
  - **$ ./prog**
    - Cria o arquivo **gmon.out**

- **Qual das funções usadas neste programa consome mais tempo de execução?**

```
#include <iostream>
#include <cmath>

#define MAX 10000

void doit()
{
   double x=0;
   for (int i=0;i<MAX;i++) x+=sin(i);
}
void f() { for (int i=0;i<1000;++i) doit();}
void g() { for (int i=0;i<5000;++i) doit();}

int main(void)
{
   double s=0;
   for(int i=0;i<1000*MAX;i++) s+=sqrt(i);
   f();
   g();
   std::cout << "Done"<< std::endl;
   return 0;
}
```

Alguma Questão?