Arquivos fonte utilizados nesta aula: <https://drive.google.com/open?id=1wZKvpF9WLXIqF6XJ_TLfaC4FySRyCghn>

Obs: a partir de agora será sempre bom **usar uma folha de papel como rascunho durante as aulas (práticas e teóricas)**

Como não tivemos assuntos novos nas aulas teóricas recentes usaremos esta prática para conhecer (de forma bem superficial) algumas ferramentas do Linux que podem ser úteis para programacao.

**Etapa 1**

Nesta aula utilizamos como estudo de caso um programa que contém duas classes muito simples: uma representa Retangulos e outra representa Circulos. Tais classes são derivadas da classe FigBase e possuem um método que, dado um ponto p (em 2D), determina se p está ou não dentro da figura (pontos exatamente na borda da figura não são considerados dentro dela).

O programa testaInclusao.cpp cria 5 figuras geométricas e conta quantas delas contém dois pontos (ponto 0,0 e ponto 15,0). Em uma folha de papel desenhe as figuras e os pontos. A seguir, veja qual deveria ter sido o resultado do programa. Conforme pode ser observado, ha bugs em algumas das classes e, portanto, os resultados impressos estão errados (não corrija os erros agora).

Uma ferramenta que pode ser útil para a localização de bugs são os debuggers ou depuradores.

Nesta aula faremos experimentos com o depurador gdb (alternativamente pode-se utilizar o front-end grafico do GDB, chamado DDD). Informações e instruções sobre o DDD e GDB podem ser obtidas nos seguintes links:

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

<https://www.gnu.org/software/gdb/>

<http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

Para depurar um programa, e’ importante compila-lo utilizando a opção -g do g++ (essa opção armazena junto ao executável do seu programa informações importantes para o depurador, como as linhas de código onde cada comando originalmente estava, nome das variáveis, etc). Após isso, utilize o comando “gdb a.out” (substituindo a.out com o nome do executável do programa).

O GDB (e outros depuradores “step-by-step”) permite que o programador acompanhe a execução do programa passo-a-passo, visualizando os valores das variáveis, o fluxo da execução, etc.

Teste o GDB com o programa fornecido como exemplo para esta aula. O comando “run” executa o programa no depurador.

Não tente modificar o código para depurá-lo (modifique-o apenas para corrigir os bugs). Embora muitas vezes a modificação de software (para, por exemplo, imprimir valores de variáveis com cout) possa auxiliar o processo de depuração, nesta aula queremos treinar apenas o uso do GDB. Dessa forma, faca toda a depuracao usando o debugger.

Explore os principais comandos do GDB junto com o professor (exemplo: break, run, next, continue, step, display, watch, finish, backtrace, etc). Após ficar familiarizado com esse debugger, use-o para tentar localizar os bugs no código que foi disponibilizado (corrija tais bugs).

Obs: ha vários tipos de debuggers. Por exemplo, um tipo particularmente útil de debugger sao os de memoria (como o Dr. Memory e o Valgrind): tais softwares são capazes de auxiliar o programador a encontrar vazamentos de memória, acesso a posições inválidas de memória (que normalmente causam falhas de segmentação), etc. É recomendado que os alunos pesquisem sobre tais debuggers, pois eles podem ser muito úteis (por exemplo em trabalhos práticos).

Links: <http://valgrind.org/docs/manual/quick-start.html> , <http://drmemory.org/>

**Etapa 2**

Agora, vamos testar um utilitario do Linux que pode ser útil em varias situacoes: o comando *time*. Tal comando pode ser utilizado para medir o tempo de execucao de algum programa ou comando.

Considere o programa “teste\_time.cpp” (disponivel junto com os arquivos desta aula pratica). Esse programa cria um vetor dinamico (da classe Vetor, que criamos durante uma das aulas teóricas) com inteiros aleatorios entre 0 e 4N-1 (onde N e’ o tamanho do vetor). A seguir, sao feitas N pesquisas de números aleatórios (tambem entre 0 e 4N-1) e o programa indica quantos de tais números estão no vetor. O primeiro argumento do programa teste\_time e’ o valor de N, enquanto o segundo argumento pode ser “b” (nesse caso, as pesquisas serao realizadas utilizando uma busca binária) ou “s” (nesse caso, as pesquisas serao feitas utilizando uma busca sequencial). A semente dos numeros aleatorios nunca e’ alterada.

Note que a complexidade da versao que realiza buscas sequenciais será O(N2) (sao feitas N buscas, onde o custo de cada busca será O(N) ) enquanto a complexidade da versao realizando busca binária será (deveria ser…) O(N log N) (o vetor será ordenado e serão realizadas N pesquisas de custo O(log N) → tempo total O(2 N log N) = O( N log N) ).

Durante este exercício crie uma tabela (igual à abaixo) para anotar o tempo de execucao do programa (em segundos) em cada experimento (células com “---” representam testes que não precisam ser feitos).

1 - Primeiramente, compile o programa com o comando “g++ teste\_time.cpp” e meca o tempo para realizar 20000 buscas binarias (use o comando “time ./a.out 20000 b”). O tempo “real” usado pelo seu programa será exibido em frente a “real” (se quiser saber sobre o que os outros valores significam pergunte ao professor).

2 - Agora, ative a otimizacao do compilador (“g++ -O3 teste\_time.cpp”) e refaca o teste anterior considerando tanto a busca binária quanto a sequencial.

3 - Refaca o teste 2 agora para N=40000.

4 - Note que mesmo para um valores “medios” de N (40000) a busca binária ficou bem mais lenta do que a sequencial!!! Algo deve estar errado… Troque os parâmetros das funcoes para que o vetor seja passado por const/referencia e refaça o teste 3.

* Quantas vezes mais rápida a busca binária ficou (em relação ao tempo do teste 3) ?
* Por que não houve alteração significativa no tempo da busca sequencial?
* Obs: dependendo do compilador e’ possivel que ele faça algumas otimizações (como colocar const/referência automaticamente em algumas situações). Porém, isso não e’ garantido e todo bom programador deve utilizar corretamente os recursos da linguagem para obter performance.

5 - Agora mude o valor de N para 80,000, 160,000 e 1,600,000 (não realize o teste com 1.6 milhao usando a busca sequencial). Como o tempo da busca binária cresce? E o da busca sequencial?

|  |  |  |
| --- | --- | --- |
| Teste | Busca binária (tempo em s) | Busca sequencial (tempo em s) |
| 1 (O0, 20,000) |  | --- |
| 2 (O3, 20,000) |  |  |
| 3 (O3, 40,000) |  |  |
| 4 (O3, 40,000, corrigido) |  |  |
| 5 (O3, 80,000, corrigido) |  |  |
| 5 (O3, 160,000, corrigido) |  |  |
| 5 (O3, 1,600,000, corrigido) |  | --- |

Discuta os resultados com o professor (esta etapa da aula não precisa ser entregue).

Observações:

* O uso de passagem de valor por copia na versao com busca binária muda a complexidade do algoritmo para O(N2 log N) (por que?). No caso da busca sequencial a complexidade fica a mesma (de toda forma o algoritmo tem que varrer o vetor (quase) completamente para procurar o elemento).
* Note que mesmo com a versao com busca sequencial sendo quadratica foi possivel processar em tempo razoavel uma entrada de tamanho ate 40,000 (isso porque foi utilizado -O3 e porque o algoritmo implementado é extremamente simples -- ou seja, as constantes na ordem de complexidade são pequenas). Em problemas um pouco mais complicados um algoritmo quadrático normalmente consegue processar em tempo “razoável” uma entrada de tamanho até 5,000.
* Observe que houve uma enorme diferença entre um algoritmo quadratico e outro nlogaritmico (essa diferença fica cada vez maior para entradas maiores). Um algoritmo cúbico, por exemplo, gastaria um tempo MUITO (mas MUITO!!) maior para processar essas entradas. Isso mostra como a ordem de complexidade importa.
* A habilidade de desenvolver algoritmos eficientes e utilizar corretamente os recursos da linguagem para obter performance é muito importante (a não ser para quem desenvolve sistemas muito simples).
* Por fim, o “time” do Linux é uma ferramenta muito simples (apesar de útil). Há tambem uma importante classe de utilitarios mais complexos chamados de “profilers” (exemplo: gprof). Tais utilitários podem analisar a execucao do seu programa e medir varias informacoes importantes relativas a desempenho (exemplo: o tempo de execução de cada função e o número de vezes em que elas foram chamadas).

**Submissao da aula pratica:**

A solucao deve ser submetida ate as 18 horas da proxima Segunda-Feira utilizando o sistema submitty ([submitty.dpi.ufv.br](http://submitty.dpi.ufv.br)). Atualmente a submissao so pode ser realizada dentro da rede da UFV.

Envie pelo sistema Submitty todos arquivos da Etapa 1. Nesta aula sua solucao não sera avaliada por testes automaticos.