Universidade de Brasília
Departamento de Ciência da Computação
Disciplina: Métodos de Programação
Código da Disciplina: 201600

Métodos de Programação - 201600

Trabalho 3

O objetivo deste trabalho é utilizar o desenvolvimento orientado a testes (TDD) para resolver o problema de contar quantas linhas de código um programa em C++ contem. Uma linha não e contabilizada se ela contem apenas espaços ou comentários, caso contrário ela e contabilizada como linha de código. Os comentários podem ser com "//" ou "/* */" conforme os comentários são reconhecidos no C++.

1) O programa deverá ser dividido em módulos e desenvolvido em C ou C++. Devera ser feito um programa para o teste.

2) O programa e o módulo devem ser depurados utilizando o GDB.
(http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html)
(https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf)

3) Faça um Makefile utilizando o exemplo de makefile 5 dado em:
(http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/)
Não se esqueça de criar os diretórios correspondentes.

4) Utilize o padrão de codificação dado em: https://google.github.io/styleguide/cppguide.html quando ele não entrar em conflito com esta especificação. O código dever ser claro e bem comentado. O código deve ser verificado se esta de acordo com o estilo usando o cpplint (https://github.com/cpplint/cpplint).
Utilize o cpplint desde o início da codificação pois é mais fácil adaptar o código no início.

5) Para cada função, colocar uma descrição do que a função faz bem como as assertivas de entrada e de saída. Deve indicar também quais são as interfaces explícita, implícita com a devida descrição, quais são os requisitos e as hipóteses de cada função.

6) Coloque nos comentários antes das funções quais são as assertivas do **contrato na especificação.** Diga o que deve ser esperado da função cliente em relação à entrada e o que deve ser garantido pela função servidora na saída.

7) As funções devem ter finalização adequada liberando memória, fechando arquivos, etc.

8) Colocar assertivas como comentários de argumentação da corretude do código.

9) Fazer assertivas estruturais: elas definem a validade das estruturas de dados e dos estados associados a estes dados.

10) Devem ser feitas revisões no código conforme visto em sala de aula utilizando as checklists do trabalho 2. Deve ser gerado um laudo (pdf) do que passou na revisão e o que não passou. Deve ser mudado o que for possível para estar dentro do especificado na checklist. O que não for possível mudar por algum motivo forte, deve estar justificado no laudo. Ex. se adequar o código a checklist exigir mudar toda estrutura do programa, não é viável fazer isto.

11) O desenvolvimento deverá ser feito utilizando um destes frameworks de teste:

11.1) gtest (https://code.google.com/p/googletest/)
11.2) catch (https://github.com/philsquared/Catch/blob/master/docs/tutorial.md)

12) Deverá ser entregue o histórico do desenvolvimento orientado a testes feito através do github (https://github.com/)

13) Instrumente o código usando o gcov. Usando o gcov.
(http://gcc.gnu.org/onlinedocs/gcc/Gcov.html). O makefile deve ser modificado de forma incluir as flags -ftest-coverage -fprofile-arcs. Depois de rodar o executável rode gcov nomearquivo e deverá ser gerado um arquivo .gcov com anotação.
O gcov é utilizado para saber qual percentual do código é coberto pelos testes. Neste caso os testes devem cobrir pelo menos 80% do código por módulo.

14) Faça a análise estática do programa utilizando o cppcheck, corrigindo os erros apontados pela ferramenta.
Utilize cppcheck --enable=warning .
para verificar os avisos nos arquivos no diretório corrente (.)
Utilize o cppcheck sempre e desde o início da codificação pois é mais fácil eliminar os problemas logo quando eles aparecem. Devem ser corrigidos apenas problemas no código feito e não em bibliotecas utilizadas (ex. gtest, catch)

15) Deve ser gerada uma documentação do código usando o programa DoxyGen (http://www.stack.nl/~dimitri/doxygen/): O programa inteiro terá de ser documentado usando DoxyGen.

16) É interessante utilizar o Valgrind (*valgrind.org/*), embora não seja obrigatório.

17) Deve ser gerado um documento (pdf ou odt) que diz como cada função foi testada. Para cada caso deve constar:

- 17.1) Nome da função, parâmetros e significado dos parâmetros. Especificação da função
- 17.2) Para cada um dos testes em cada função
  - 17,2.1) Nome de cada teste
  - 17.2.2) O que vai ser testado
  - 17.2.3) Qual deve a ser a entrada
  - 17.2.4) Qual deve ser a saída
  - 17.2.5) Qual é o critério para passar no teste
  - 17.2.6) Se a sua função efetivamente passou no teste ou não
- 17.3) Responda em que casos a função não retorna um resultado válido.
  - 17.3.1) Existem funções que podem corromper a estrutura de dados? Como?
  - 17.3.2) O que pode ser feito para evitar este problema

Devem ser enviados para a tarefa no ead.unb.br um arquivo zip onde estão compactados todos os diretórios e arquivos necessários. O documento deve estar na raiz do diretório. Todos os arquivos devem ser enviados compactados em um único arquivo (.zip) e deve ser no formato matricula_primeiro_nome ex: 06_12345_Jose.zip. Deve conter também um arquivo leiame.txt que diga como o programa deve ser compilado.

O arquivo deverá conter um link para o github correspondente. Ele será usado para avaliar se o desenvolvimento realmente foi feito orientado a teste.

Data de entrega:

11/ 10 /18

Pela tarefa na página da disciplina no ead.unb.br