**Universidade Federal do Rio de Janeiro – Escola Politécnica**

**Rio de Janeiro, 28 de Maio de 2012**

**Prof.: Leonardo Nunes Disciplina: Linguagens de Programação**

**Alunos: Daniel Ramos Garcia Turma: EL1 + ECA**

**Fabio Oliveira Baptista da Silva**

**Fellipe Diogo Falleiro**

**Fernando de Moura Diniz do Rego Monteiro**

**Marcos Paulo Oliveira Silva**

**2º Trabalho – Calculadora em Linha de Comando**

1) Modularização do Projeto

O projeto da calculadora em linha de comando foi, inicialmente, subdividido em 5 módulos:

- Módulo Principal ( main() )

- Módulo Gerenciador

- Módulo Validador

- Módulo Conformador

- Unidade Aritmética

Onde cada módulo ficaria responsável por tarefas bem específicas, como a validação da variável de destino, “preparação” da expressão aritmética para o cálculo, o qual seria efetuado pela unidade aritmética, e um módulo que gerenciasse todo este processo.

Após algumas considerações apresentadas pelo grupo, foi verificada a necessidade da implementação de mais 2 módulos auxiliares:

- Módulo Armazenador (Memória)

- Módulo Tratador de Erros

Nesta fase de desenvolvimento foi decidido os responsáveis pela implementação de cada módulo, além de uma visão geral do que cada módulo deveria executar, e de que maneira e com quais módulos se comunicaria.

2- Os Módulos

a) Módulo Principal (Responsáveis: Daniel e Fernando)

O módulo principal tem como objetivo interagir com o usuário do programa. Inicialmente, o programa exibe uma mensagem informando as funções e espera um comando. Ao receber uma expressão, o programa avalia se foi: uma expressão propriamente dita ou um comando pré-definido.

Comandos esses que são:

- *all*: Chama a função exibir memória do bloco de memória;

- *help*: Exibe como o usuário deve utilizar o programa;

- *quit*: Quebra o *loop*, fechando o programa.

Caso nenhum desses comandos seja inserido, a função *setString* do Gerenciador é chamada.

Ocorrendo algum erro dentro de qualquer um dos módulos, esse erro é ‘lançado’ usando o *throw* e o módulo principal usa o *catch* e a função *getErro* para retornar o erro. Não ocorrendo nenhum erro, a variável é exibida junto ao seu valor correspondente e o programa espera novamente um comando vindo do usuário.

b) Módulo Gerenciador (Responsáveis: Daniel e Fernando)

O Gerenciador é o módulo central do programa. Ele recebe a entrada inicial, separa as atribuições caso haja mais de uma na mesma entrada e as repassa entre os módulos, gerando uma *string* com todas as variáveis e seus respectivos valores e a retorna para o programa principal.

b.1)std::string setString(string entrada);

A função recebe a expressão vinda do módulo principal e a define como a variável entrada. Em seguida procura dentro da variável pelo “;”,definido no *namespace* Gramática, pois é nesse ponto onde o programa faz a quebra de linha. Uma a uma, elas são retiradas da *string* e enviadas para o Validador por referência usando a função *setString* do Validador.

Após passar pelo Validador, o Gerenciador pega o nome da variável usando a função *getName*, a *string* modificada é então passada para o Conformador que a modifica novamente, preparando-a para a conta. Em seguida é passada para a Unidade Aritmética que retorna o valor da conta resolvida. Esse valor e o nome são então inseridos na memória através da função *insere*, e este resultado é utilizado junto do nome da variável, na construção da *string* que será impressa e que é retornada no fim do módulo.

c) Módulo Validador (Responsáveis: Daniel e Fernando)

O objetivo do Validador é avaliar se a expressão foi construída corretamente e separar o nome da variável e a conta atribuída a ela.

c.1)void setString(string &linha);

Antes de qualquer verificação, todos os espaços são retirados da string passada. Posteriormente, inicia-se uma busca pelo sinal de atribuição “:=”, definido no *namespace* Gramática. Em seguida, ocorre a separação da *string* e inicia-se uma bateria de testes. Utilizando as definições de variáveis válidas contidas na gramática, o Validador testa se o nome da variável é válido, posteriormente avalia se não foram usados caracteres inválidos na conta. Por fim, a função atribui a *string* passada por referência, apenas a *string* com a conta.

c.2)std::string getName ();

A função *getName* definida no módulo, retorna o nome da variável.

d) Módulo Conformador (Responsável: Fellipe )

O objetivo principal desse módulo é ajeitar a expressão recebida do gerenciador para prepará-la para a Unidade Aritmética. Antes de chegar no conformador, essa expressão foi validada quanto a existência de caracteres inválidos.

Para implementar o conformador.cpp, foi necessário criar duas funções auxiliares.

. string doubleToString (double number)

. unsigned checkField (string myset,char mychar)

A primeira função, a *doubleToString*, como o nome já diz, ela irá receber um valor numérico do tipo *double*, que será o *number*, e irá retornar esse *double* convertido como uma string. A implementação da *doubleToString* foi feita de maneira simples, através da biblioteca *sstring*.

A segunda função foi muito útil durante a implementação do conformador.cpp, pois podemos usá-la para verificar se o caractere *mychar* está presente dentro da string *myset*. Usando essa ideia ficou simples descobrir de qual elemento da gramática se tratava do caractere em questão. A função retornava 1 caso o caractere existisse, e se não existisse, o código de retorno era 0.

A eliminação dos espaços foi importante, feito no Validador, foi importante pois nos dava a certeza que todos os caracteres possuíam como vizinhos, caracteres que estavam definidos no nosso *namespace* Gramática. Porém também seria necessário verificar o primeiro e o último caractere da expressão, pois essas duas posições possuem regras especiais.

Como já era sabido que a *string* não possuía espaços em branco, a validação foi feita verificando cada par de caracteres adjacentes da *string*. Essa verificação foi feita usando a função *checkField*, pois através da *checkField* era possível definir, de acordo com o caractere que estamos considerando, qual seria o campo de caracteres inválidos para ser o seu “vizinho” posterior.

Após validar a expressão, o próximo objetivo do conformador era substituir as variáveis por seu valor numérico correspondente. Para realizar a substituição, foi necessário usar o método busca do módulo de memória. A existência da variável, ou não, foi importante nesse momento, pois se não existisse, estaríamos lidando com uma expressão inválida.

Após validar a expressão e substituirmos os valores das variáveis, essa expressão já estava pronta para ser usada na nossa unidade aritmética.

e) Unidade Aritmética (Responsáveis: Fábio e Marcos)

A unidade aritmética foi implementada seguindo as expressões regulares discriminadas no roteiro do trabalho, observando as regras de associatividade e precedência existentes na aritmética.

Esta unidade possui três funções que trabalham associadas, cada uma realizando uma tarefa específica:

e.1) double calcular (std::string expressão);

Esta função tem por finalidade encontrar pares de parênteses e enviar para a função *calcularExpressão*, para a realização do cálculo, o conteúdo destes. Vislumbrando a possibilidade de se encontrar parênteses aninhados, a função se utiliza da técnica de recursividade para chamar ela mesma, até que seja encontrado o par de parênteses mais interno, e então enviar para a função *calcularExpressão*, a qual devolve um valor *double* que é transformado em *string* por um conversor *stringstream* e em seguida inserido de volta *string* original. Este processo é repetido até que não se encontre mais nenhum par de parênteses, sendo a *string* resultante sendo enviada mais uma vez para a função *calcularExpressão*. Quando o módulo realiza uma substituição do par de parênteses por um número negativo, para evitar que se torne uma expressão com dois operadores juntos, foi utilizado um caractere especial "!", para indicar que o próximo elemento é negativo. Isso é tratado na função calcularTermo.

e.2) double calcularExpressao ( std::string &expressao );

Esta função tem por finalidade encontrar os operadores ‘+’ e ‘-‘ , e associar os operandos pela esquerda. Esta tarefa é implementada enviando, primeiramente, o termo à direita do operador mais à direita, para a função *calcularTermo*,a qual devolve o resultado, e recursivamente enviar para outra instância desta mesma função a expressão à esquerda deste mesmo operador. Repetindo o processo até obter a expressão final, apenas com os operadores ‘+’ e ‘-‘, para calculá-los, e retornar o resultado.

e.3) double calcularTermo ( std::string &termo );

Esta função tem por finalidade encontrar os operadores ‘\*’ e “\”, e associar os operandos pela esquerda. Esta tarefa é implementada calculando, primeiramente, o elemento à direita do operador mais à direita, e recursivamente enviando para outra instância desta mesma função a expressão à esquerda deste mesmo operador. Repetindo o processo até obter a expressão final, apenas com os operadores ‘\*’ e ‘/‘, para calculá-los e retornar o resultado.

Foi decidido neste módulo que ele trataria os erros de divisão por zero, e parênteses sem conteúdo. Além, dos sinais representativos das operações. E o caractere especial "!", indicador de um número negativo que estava entre parênteses.

f) Módulo Armazenador (Responsáveis: Fábio e Marcos)

Este módulo foi implementado para possibilitar o armazenamento da variável e seu valor na memória, para um possível resgate de seu valor.

Foi decidido que em caso da variável já existir na memória, que seu valor seria sobrescrito.

Este módulo conta com os seguintes componentes:

f.1) struct celula

{

std::string identificador;

double valor;

};

std::list<celula> ram;

Foi decidido utilizar uma lista como estrutura de armazenamento devido aos métodos pré-existentes para esta classe, que facilitam sua manipulação.

f.2) void insere ( std::string nome, double dado);

Este método é responsável pela inserção de uma nova variável na lista. Com o auxílio de um iterador, fazemos uma busca na lista, caso a variável já exista, ela é sobrescrita, caso não exista, ocorre uma inserção no fim da lista.

f.3) double busca ( std::string nome );

Este método realiza a busca de um dado na lista. Novamente com o auxílio do iterador, efetuamos a busca e retornamos o valor da variável buscada. Caso, a variável buscada não exista é gerado um erro.

f.4) void exibirMemoria ( void );

Este método é responsável por exibir o conteúdo de toda a memória quando o usuário escreve *all* na linha de comando.

Ao realizar um *loop*, percorre toda a lista, e mostra ao usuário todas as variáveis já declaradas, relacionando nome da variável e o valor da mesma.

g) Módulo Tratador de Erros (Responsáveis: Fábio e Marcos)

Este módulo tem por finalidade tratar os erros lançados pelos módulos de forma a informar ao usuário, que tipo de erro ocorreu. Para isso foi criado um tipo enumerado contendo os todos os erros possíveis e uma função *getErro* para tratá-lo.

g.1 typedef enum

{

// Erros da Memoria

VARIAVEL\_NAO\_EXISTENTE,

// Erros Gerenciador

// Erros Validador

VARIAVEL\_INVALIDA,

SEM\_ATRIBUICAO,

SEM\_EXPRESSAO,

EXPRESSAO\_INVALIDA,

// Erros Conformador

PRIMEIRO\_CARACTERE\_INVALIDO,

ULTIMO\_CARACTERE\_INVALIDO,

OPERADORES\_JUNTOS,

NUMERO\_SEPARADORES,

USO\_INADEQUADO\_DO\_SEPARADOR,

AUSENCIA\_OPERADOR,

USO\_INADEQUADO\_DO\_PONTO,

// Erros da Unidade\_aritmetica

DIVISAO\_POR\_ZERO,

SEPARADOR\_SEM\_CONTEUDO,

} tipoErro;

g.2) std::string getError ( tipoErro erro );

Esta função recebe um tipoErro, e dependendo do erro recebido, joga na tela uma *string* explicando ao usuário, o que ocorreu.

3- Conclusões e Considerações Finais

A calculadora, desde o início, foi projetada para que cada módulo tivesse tarefas bem definidas, o que facilitou a visualização de como seria sua interface e com quais módulos haveria comunicação. Porém, durante o processo de implementação houve a necessidade de uma reorganização de algumas tarefas.

Por exemplo, a retirada de espaços da *string* que contem a expressão, a princípio, ficava a cargo do Conformador, efetuar esta tarefa, mas devido aos espaços ainda não retirados, a implementação do Validador ficaria bem mais trabalhosa. Então, foi decidido que esta tarefa seria realizada pelo Validador, e não mais pelo Conformador.

A decisão de projeto de ligar sempre com a *string* foi a que gerou mais problemas, principalmente no que tange a substituição de um determinada valor na *string*.

Outras decisões também tiveram de ser contornadas, por exemplo, na Unidade aritmética, a função *calcularExpressao*, se baseava apenas na aparição dos operadores de soma e subtração, o que impedia que uma expressão tivesse um operador de multiplicação ou divisão junto de um caractere de soma ou subtração. Essa restrição forçou uma mudança tanto na própria Unidade Aritmética como no Conformador.

A Unidade Aritmética não poderia substituir uma expressão entre parênteses por um número negativo, forçando a modificação já comentada do caractere “!”. O Conformador por sua vez, passou a considerar uma expressão com abre parênteses seguida de uma sinal de soma ou subtração como errada, sendo forçada a inserir um 0 complementar, para que a unidade aritmética fizesse a conta corretamente.

Após a finalização de todo o projeto, também foi pensado que talvez tivesse sido melhor outra modularização do problema. Por exemplo, se o módulo Validador, ficasse responsável por toda a validação da expressão, e não só pela da variável em si, poder-se-ia tornar o módulo Conformador (sua função de buscar valor da memória) como uma função da própria Unidade Aritmética, a fim de evitar a substituição na *string*. Além de uma modificação na forma de implementação da Unidade Aritmética, para torná-la mais maleável, diminuindo o número de exigências da expressão, por exemplo, declarando variáveis temporárias com valor equivalente ao valor dos parênteses para que a substituição na *string*, não forçasse nem o truncamento, nem o problema do número negativo.

Pode-se verificar que as decisões tomadas na fase de projeto, implicaram em diversas complicações na fase implementação. Portanto, poder-se-ia ter gasto mais tempo nas especificações dos módulos do projeto, de forma a evitar complicações desnecessárias e tornar o código mais compreensível, compacto e eficiente.