UNIVERSIDADE DE BRASÍLIA

DISCIPLINA: Métodos de Programação

Trabalho 2 - *checklist* para verificação de programas C++

ALUNO: EDUARDO MARQUES DOS REIS 19/0012358

1. Como Fazer o Levantamento de Requisitos

1.1. Organização e Completeza

1.1.1. Todas as referências cruzadas estão corretas?

1.1.2. O requisito está em um nível apropriado de detalhe?

Ex: Sem o excesso de informações desnecessárias, mas também não deixando de contém informações importantes

1.1.3. Seguir algum de codificação e aplicá-lo em todo o código

Ex: Por exemplo utilizar o modelo de codificação pré-estabelecido em: https://google.github.io/styleguide/cppguide.html

1.2. Integridade

1.2.1. O comportamento esperado está documentado para todas as condições de erro previstas?

Ex: Através da documentação verificar se o código está de acordo com o esperado

1.2.2. Está faltando alguma informação necessária em um requisito?

Ex: Os testes devem abranger todos os requisitos necessários para um bom funcionamento do sistema desenvolvido

1.2.3. Todas as necessidades dos clientes estão sendo atendidas?

Ex: em diálogo com o cliente, deve perceber todas os requisitos que o cliente deseja ver na aplicação final.

1.3. Exatidão

1.3.1. Algum requisito entra em conflito ou duplica outros requisitos?

Ex: em diálogo com o cliente, deve perceber todas os requisitos que o cliente deseja ver na aplicação final.

1.3.2. Cada requisito está escrito em linguagem clara, concisa e inequívoca?

Ex: Em um sistema com o modelo de programação orientado a testes, todos os testes devem estar sendo alcançados para continuar o progresso da aplicação desenvolvida, e não esperar muitos erros para começar a resolver

1.3.3. Cada requisito pode ser verificado por meio de teste, demonstração, revisão ou análise?

Ex: Em um sistema com o modelo de programação orientado a testes, todos os testes devem estar sendo alcançados para continuar o progresso da aplicação desenvolvida, e não esperar muitos erros para começar a resolver

1.4. Atributos de qualidade (requisitos não funcionais)

1.4.1. Todos os objetivos de desempenho estão devidamente especificados?

Ex: Por meio de testes verificar se os todos os objetivos que são requeridos estão sendo desempenhado corretamente

1.4.2. Todas as considerações de proteção e segurança estão especificadas de maneira adequada?

Ex: Por meio de testes verificar se não a falhas se segurança no código, e verificar se não é possível, a invasão de terceiros

1.5. Rastreabilidade

1.5.1. As dependências de requisitos estão sendo identificadas?

Ex: Se um requisito depende de outro, ambos devem funcionar corretamente independentemente da situação

1.5.2. Cada requisito é rastreável até sua origem?

Ex: Se um requisito depende de outro, como herança, deve ser possível rastrear algum método até sua origem

1.6. Disponibilidade do código

1.6.1. Os códigos presentes no sistema estão divididos em suas respectivas áreas. Como funções em arquivos .hpp e métodos dessas funções em arquivos .cpp

Ex: separar em arquivos de cabeçario (.hpp ou .p), classes e métodos inlínes, e construir um arquivo (.cpp ou .c) para o corpo dos métodos iniciado no cabeçario, e ainda um outro arquivo para a função principal (main.cpp)

Ex: não limitar em apenas 3 arquivos. Posso criar um par de arquivos de acordo com minha necessidade, como arquivos (.cpp e .hpp) para entidades, produtos, testes, ...

1.7. Comentários

1.7.1. Produzir de forma sucinta e necessária comentários em partes do código na qual não é possível a compreensão do mesmo apenas pela leitura do código

Ex: valor = 1.4 * prazo + recesso

// valor é o resultado final do dinheiro que devo ao banco

// 1.4 é a taxa anual oferecida pelos juros

1.7.2. Documentar por meio de ferramentas (como doxygen) todas as partes principais do código, como as classes e testes realizados

Ex: documentação para a classe aplicação

/**,

* @brief Entidade para Aplicação

*

*/

1.8. Uso de armazenamento

1.8.1. As matrizes e variáveis são grandes o suficiente?

Ex: Escolher o melhor tipo de armazenamento de acordo com a necessidade do programa

1.9. Entrada-saída

1.9.1. Todas as exceções foram tratadas?

Ex: Verificar se em todas as exceções que o programa pode apresentar, se existe um tratamento adequado

1.9.2. Cada requisito está livre de erros de conteúdo e gramaticais?

Ex: Por meio de verificação manual verificar se o código possui erros gramaticais

1.10. Edições especiais

1.10.1. Os requisitos fornecem uma base adequada para projeto e teste?

Ex: conseguir fazer todos os requisitos funcionais através dos testes pré estabelecidos contendo todas as funcionalidades do sistema

1.10.2. Todos os requisitos são realmente requisitos, e não soluções de design ou implementação?

Ex: Fazer todo o código baseando em os requisitos estarem funcionando corretamente e não apenas passar nos testes

2. Como fazer o design do software

2.1. Construtores

2.1.1. Funções que serão chamadas sempre em uma classe podem ser construtores da mesma

Ex: Uma classe pode incluir uma função especial chamada seu construtor, que é automaticamente chamada sempre que um novo objeto dessa classe é criado, permitindo que a classe inicialize variáveis de membro ou aloque armazenamento

```
class Lista {
        int valor, posicao;
public:
        Lista (int,int);
        int area () {return (valor*posicao);}
};
```

2.2. Destrutores

2.2.1. Liberar recursos obtidos por construtores

Ex: Destrutores realizam a função inversa: são funções invocadas quando um objeto está para ``morrer''. Caso um objeto tenha recursos alocados, destrutores devem liberar tais recursos. Por exemplo, se o construtor de uma classe alocou uma variável dinamicamente com new, o destrutor correspondente deve liberar o espaço ocupado por esta variável com o operador delete.

```
        Lista (int, int);
        ~Lista ();      // método destrutor da classe Lista
```

2.3. Sobrecarga de Construtores

2.3.1. Cada construtor pode ser diferenciado pelo nome ou pelo número de entradas.

Ex: Lista();

Lista (int, int);

O compilador irá chamar automaticamente aquele cujos parâmetros correspondem aos argumentos

2.4. Inicialização Uniforme

2.4.1. Ao invez da inicialização funcional citada acima (2.3.1), posso utilizar a inicialização uniforme

Ex: Lista valor1 = 20.0; // assignment init.

Lista valor2 {30.0}; // uniform init.

Uma vantagem da inicialização uniforme sobre a forma funcional é que, ao contrário dos parênteses, as chaves não podem ser confundidas com as declarações de função e, portanto, podem ser usadas para chamar explicitamente os construtores padrão

2.5. Inicialização de membro em construtores

2.5.1. Quando um construtor é usado para inicializar outros membros, esses outros membros podem ser inicializados diretamente, sem recorrer a instruções em seu corpo.

Ex: Isso é feito inserindo, antes do corpo do construtor, dois pontos (:)

Lista::Lista (int x, int y) {valor = x; posicao = y;}

Ou ainda, pode-se fazer (utilizando inicialização de membros)**:**

Lista::Lista (int x, int y): valor(x), posicao(y) { }

2.6. Ponteiro para classes

2.6.1. Os objetos também podem ser apontados por ponteiros: uma vez declarada, uma classe se torna um tipo válido, portanto, pode ser usada como o tipo apontado por um ponteiro.

Ex: Lista * ptrLista

*x = apontado por x

&x = endereço de x

2.7. Classes definidas com struct

2.7.1. As classes podem ser definidas não apenas com palavras-chave CLASS, mas também com palavras-chave STRUCT.

Ex: A palavra-chave STRUCT, geralmente usada para declarar estruturas de dados simples. A única diferença entre o CALSS e STRUCT é que STRUCT tem o

acesso por padrão como PUBLIC, já na classe você pode definir como PUBLIC, PRIVATE ou PROTECT

2.8. Funções vazias

2.8.1. Se a função não precisar retornar um valor?

Ex: void imprimir (){

cout << "Sou uma função!";

}

Posso passar um void como parâmetro, para especificar que a função não aceita parâmetros

Void imprimir(void)

2.9. Retorno do valor principal

2.9.1. Preciso retornar algo se minha função for diferente de void

Ex: Todas as função que não retornam 'vazio', precisam de um return mostrando que a função acabou ali e que tenho algo para retornar, por exemplo na int main():

Return 0;

2.10. Argumentos passados para funções

2.10.1. Posso passar os argumentos como valor

Ex: void adicionar (int a, int b)

adicionar (x, y) // na main

Os valores passados para a função faram o que a função mandar, porém quando voltar para a função principal (ou o local aonde foi chamado a função adicionar), os valores de a e b não serão alterados

2.10.2. Posso passar os argumentos como referência

Ex: void duplicar (int& a, int& b){

a*=2; b*=2;

}

duplicar (x, y); // na main

Os valores passados de a e b serão alterados e continuaram assim mesmo quando voltar para função principal

2.11. Referencias constantes

2.11.1. Para parâmetros compostos que podem ser contem grande quantidade de bit's, passar por referência pode ser a melhor opção.

Ex: string concatenar (string& a, string& b){

return a+b;

}

Ao invés de

string concatenar (string a, string b){

return a+b;

}

2.12. Funções inline

2.12.1. Chamar uma função geralmente causa uma certa sobrecarga (empilhar argumentos, saltos, etc ...), por isso, se a função for curta, fazer ela inline pode aumentar o desempenho do projeto

Ex: inline string concatenar (const string& a, const string& b) {

return a+b;

}

2.13. Valores padrões em parâmetros

2.13.1. As funções também podem ter parâmetros opcionais, para os quais não são necessários argumentos na chamada

Ex: int dividir (int a, int b=2){

int r; r=a/b;

return (r);

}

2.14. Declarando funções

2.14.1. As funções não podem ser chamadas antes de serem declaradas.

Ex: void odd (int x);

// posteriormente vem a função main, e depois:

```
void odd (int x){
 if ((x%2)!=0)
 cout << "It is odd.\n";
 else even (x);
}
```

2.15. Recursividade

2.15.1. Posso utilizar da recursividade para reduzir a quantidade de códigos digitados, ou ainda, utilizar para funções matematicas, como o fatorial

Ex: int factorial (long a){

```
 if (a > 1)   return (a * factorial (a-1));
 else   return 1;
}
```

A função vai chamar ela mesma, até que a condição de parada seja atendida

2.16. Defeito de definição de classe

2.16.1. Cada classe tem construtores e destruidores apropriados?

Ex: Colocar em uma classe apenas construtores e destrutores realmente necessários

2.16.2. Alguma subclasse possui membros comuns que deveriam estar na superclasse?

Ex: caso um membro em comum que esteja em uma subclasse (exemplo: em uma conta de banco, todos usuários tem o código do banco, que é comum para todos os usuários), colocar em uma classe mãe

2.16.3. A hierarquia de herança de classe pode ser simplificada?

Ex: não criar hierarquias que compliquem o código, e realmente fazer hierarquia de classe caso for necessário (exemplo: em uma conta de banco eu poderia ter dois tipos de classe, conta corrente e conta poupança, ao invez de uma só com as duas características)

2.17. Defeitos computacionais e numéricos

2.17.1. É possível overflow ou underflow durante um cálculo?

Ex: evitar qualquer possibilidade que isso ocorra (exemplo: se o usuário digitar 1, eu entrar em um loop infinito, evitar de digitar esse algarismo)

2.17.2. Os parênteses são usados para evitar ambiguidades?

Ex: evitar erros de interpretação no código, fazer:

```
If (((A == B) < C) > D)
```

Ao invés de

```
If (A == B < C > D)
```

2.18. Controle de fluxo

2.18.1. Para cada loop: A melhor escolha de construções em loop é usada?

Ex: quando eu tiver uma quantidade certa de quantos loop's vou utilizar, fazer isso utilizando o for e não o while, para evitar error

```
For(i = 0; i > 2; i++){
}
```

Ao invés de

```
Do{
}While(true);
```

2.18.2. Todos os loops serão encerrados?

Ex: Lembrar de fechar todos os loop's para não fazer o programa repetir uma parte do código infinitas vezes (causando a falha do sistema)

2.18.3. Todas as exceções são tratadas de forma adequada?

Ex: try{,

(código)

}

catch(invalid_argument &excecao){

(código)

}

2.19. Controle de modularidade

2.19.1. Existe código repetitivo que poderia ser substituído por uma chamada a um método que fornece o comportamento do código repetitivo?

Ex: Lembrar de não repetir código (criação de funções), e refatorar códigos na qual existe um código muito repetitivo

2.19.2. As bibliotecas de classes são usadas onde e quando apropriado?

Ex: lembrar que possuem bibliotecas prontas feitas por organizações ou usuários que podem facilitar a codificação, e verificar se são a melhor opção (pode ser que uma biblioteca ajuda na codificação, mas pode causar outro problema)

2.20. Expectativa de desempenho

2.20.1. Podem ser usados melhores estruturas de dados ou algoritmos mais eficientes?

Ex: Se for utilizar um algoritmo ordenação, verificar se ele é o melhor para o caso requerido (existem diversos algoritmos, alguns são mais eficientes com muitos dados, outros até uma certa quantia de dados são melhores)

2.20.2. Todos os resultados calculados e armazenados são realmente usados?

Ex: não criar, variáveis, funções classes, testes ou qualquer outra coisa na qual não vai ser utilizada

Bastante desses métodos foram inspirados de documentação online em: http://www.cplusplus.com/doc/

3. Como fazer a codificação

3.1. Declaração de Variáveis

3.1.1. Os Arrays estão declarados com especificação ao invés de números?

Ex: int mes[quantidadeMes]

Ao invés de

Int mês [12]

3.1.2. Caso o valor de uma variável não for mudar, defini-la como constante pode ser a melhor opção para evitar erros

Ex: const unsigned mês = 12;

Ao invés de

Int mês = 12;

3.1.3. Ao invés de definir algo no sistema posso utilizar as constantes novamente

Ex: const unsigned numero = 10

Ao invés de

#define numero 10

3.2. Condicionais

3.2.1. Utilização de condições quando de número de ponto flutuantes, é melhor utilizar um maior ou menor de acordo com suas necessidades pois o sistema pode arredondar.

Ex: if (soma <= 0.1)

Ao invés de

If (soma == 0.1)

3.3. Inicialização

3.3.1. Não esquecer de iniciar suas variáveis locais antes de utilizar, e se possível inicializar como vazio (bom fazer isso em C)

Ex: int contador = 0;

contador++;

3.4. Dimensionamento de dados

3.4.1. Cuidado ao utilizar argumentos para o sizeof, mesmo não estando incorretas, não é uma boa prática de programação.

Ex: sizeof (ptr)

Ao invés de

sizeof(* ptr)

Ex2: sizeof (array)

Ao invés de

sizeof (array [0])

3.5. Controle de overflow

3.5.1. Não permitir que em um loop o limite inferior seja um limite exclusivo

Ex: cuidado ao fazer o exemplo abaixo:

```
int a = 0;
do{
    a++;
}while(a < 1)
```

Pois entra em um loop infinito

3.6. Alocação de dados dinâmicos

3.6.1. Quando não se sabe exatamente a o tamanho de sua variável, a alocação dinâmica resolve por não reservar um valor limite máximo.

Ex: vetor = (int *) malloc (100*sizeof(int));

3.7. Desalocação de dados dinâmicos

3.7.1. Vetores estão sendo excluídos como se fossem escalares?

Ex: delete [] meuArray;

Ao invés de

Delete meuArray;

3.7.2. Não é necessário excluir armazenamentos já excluídos.

3.8. Ponteiros

3.8.1. O armazenamento excluído ainda tem ponteiro para ele? É uma boa pratica de programação definir ponteiros como NULL após a exclusão;

Ex: free(aux);

aux = (struct ptr*) NULL;

Ao invés de apenas

Free(aux);

3.9. Argumento como parâmetros

3.9.1. Certificar que os parâmetros passados não excedem o limite da função

Ex: de uma função que são passados um inteiro e uma string

void soma(int *a, int *b)

3.10. Retornar Valores

3.10.1. Funções que retornam um certo tipo de valor deve receber esse tipo de valor de retorno.

Ex: Uma função que retorna inteiro

int valor;

valor = funcao();