

PONTIFÍCIA U
NIVERSIDADE C
ATÓLICA
DO RIO DE JANEIRO



Pontifícia Universidade Católica do Rio de Janeiro

INF1301 Caderno
Programação Modular

Alunos 1811208 - Suemy Inagaki
Professor Flávio Heleno Bevilacqua e Silva

Rio de Janeiro, 13 de Maio de 2019

Conteúdo

1 Aula 01 - 13/03/2019	1
1.1 Introdução	1
1.2 Pincípios de Modularidade	2
2 Aula 02 - 18/03/2019	4
2.1 Interface Fornecida por Terceiros	4
2.2 Interface em Detalhe	5
2.3 Exemplo de Interface	5
2.4 Processo de Desenvolvimento	6
2.5 Bibliotecas Estáticas e Dinâmicas	7
2.5.1 Estática	7
2.5.2 Dinâmica:	7
2.6 Módulo de definição (.h)	7
2.7 Módulo de Implementação (.c)	7
2.8 Tipo Abstrato de Dados (TAD)	7
3 Aula 03 - 20/03/2019	8
3.1 Conceito	8
3.2 Singleton	8
3.3 Normal	9
4 Aula 04 - 25/03/2019	9
4.1 Propriedade da Modularização	9
4.2 Encapsulamento	10
4.3 Acoplamento	10
4.4 Coesão	11
5 Aula 05 - 27/03/2019	11
6 Aula 06 - 01/04/2019	11
7 Aula 07 - 03/04/2019	11
7.1 Requisitos	12
7.1.1 O que são Requisitos	12
7.1.2 Característica dos Requisitos	12
7.1.3 Etapas da Especificação	12
7.1.4 Tipos de Requisitos	12
7.1.5 Exemplos de Requisitos	12
7.1.6 Vetor	12
7.1.7 Lista Simplesmente Encadeada Com Cabeça	12

7.1.8	Lista Duplamente Encadeada Com Cabeça	13
7.1.9	Arvore Binaria Generica Com Cabeça	13
8	Aula 08 - 08/04/2019	14
8.1	Assertivas Estruturais	14
9	Aula 09 - 15/04/2019	14
9.1	Vetor de Lista de arvores	14
10	Aula 10 - 17/04/2019	14
10.1	Assertivas	14
10.1.1	Assertivas Estruturais	14
10.1.2	Assertivas de Entrada e Saída	14
11	Aula 11 - 22/04/2019	15
12	Aula 12 - 24/04/2019	15
12.1	Implementação da Programação Modular	15
12.1.1	Espaço de Dados	15
12.2	Tipo de dados	15
12.3	Tipos Básicos	15
13	Aula 13 - 29/04/2019	15
13.1	Pré-processamento	15
13.2	Exercicios da Lista	15
14	Aula 14 - 06/05/2019	15
14.1	Paradigma	16
14.2	Estrutura de Funções	16
14.3	Estrutura de Chamadas	16
14.4	Função	16
14.5	Especificação de Função	16
14.6	House Keeping	16
15	Aula 15 - 08/05/2019	16
15.1	Repetições	16
15.2	Recursão	16
15.3	Estado	16
15.4	Esquema de Algoritmo	16
15.5	Parâmetros do tipo ponteiro para função	16

1 Aula 01 - 13/03/2019

1.1 Introdução

Vantagens da Programação Modular

1. Dividir o problema em subprobleminhas
2. Cada pessoa trabalhando em um sub problema evita barreira de complexidade
3. Distribuir tarefas em um grupo (*paralelizar*)
4. A dificuldade é a fase de análise, definindo os módulos necessários para o trabalho
5. Reuso dos módulos *especializar um assunto agiliza!!!* Ver *Grafico 1*
6. Criação de um acervo de módulos reutilizados dentro de um nicho de negócios
7. Permite trabalhar com *baselines* de módulos já testados. *se precisar de um upgrade e der algum erro, é possível voltar à versão .exe antiga* Ver *Grafico 2*
8. Desenvolvimento incremental: Criou um módulo, testa. Não precisa testar só quando terminar todos os módulos. Se um outro módulo necessário para testar não tiver pronto, criar uma chamada FAKE
9. Aprimoramento individual
10. Reduz tempo de compilação (só precisa compilar tudo uma unica vez)

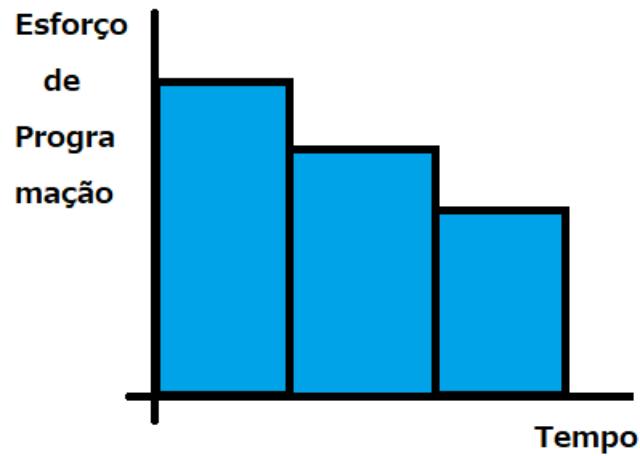


Figura 1:

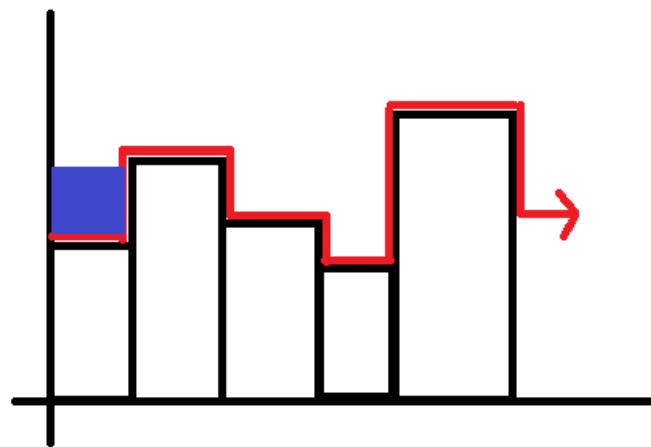


Figura 2:

1.2 Pincípios de Modularidade

Obs: Módulo de baixa qualidade é o módulo com mais de um conceito.
Módulo

1. Física: unidade de compilação independente.
2. Lógica: possui um único conceito ou objeto. *O que o módulo faz? A resposta tem que ser única.*

Elemento de Programas Podem ser: blocos de códigos, fragmentos de textos de documentação, funções, figuras de diagramas, seções de documentações, tipo de dados, classes, componentes...

Construto (Ou build): É a versão de uma aplicação que pode ser executada mesmo que incompleta.

Artefato: Algo elaborado durante um processo de desenvolvimento e que possui identidade própria. É o que pode ser versionado em uma aplicação.

Pergunta: Qual a relação entre interface e módulo? Hierarquia: (Ver Figura 3)



Figura 3:

Interface É o mecanismo de troca de dados, comandos e eventos entre elementos de programas.

Comando: "Grave um arquivo" Evento: "sem conexão", "arquivo vazio"

Obs: A interface sempre ocorre entre elemento da programação de mesmo nível na hierarquia.



Figura 4:

Formas de Interface:

1. Entre sistemas: Comunicar-se por arquivos. Um sistema lê o arquivo do outro.
2. Entre Módulos: Funções. Um módulo chama a função do outro. Funções de acesso e seus parâmetros.
3. Entre Blocos de Códigos: variáveis globais aos blocos (Interface entre linhas)
4. Relacionamento Cliente-Servidor *Caso especial: Callback*. *M1 não fornece dado suficiente, então M2 chama M1 solicitando mais dados, depois disso, M1 chama M2 de novo. Então M1 e M2 trocam de papéis temporariamente Ver Figura 6*

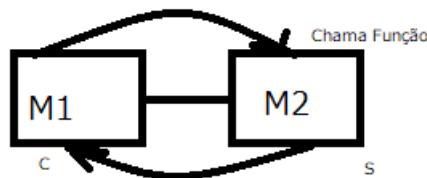


Figura 5:

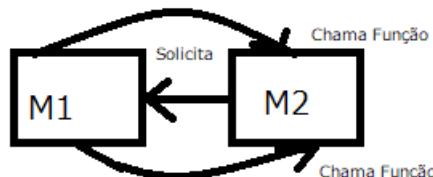


Figura 6:

2 Aula 02 - 18/03/2019

2.1 Interface Fornecida por Terceiros

É quando um tipo utilizado em uma interface entre dois módulos não está definida em nenhum dos módulos de implementação e sim no módulo de definição comum aos dois.

Nunca duplicar código: você pode acabar fazendo manutenção em um e esquecer do outro



Figura 7:

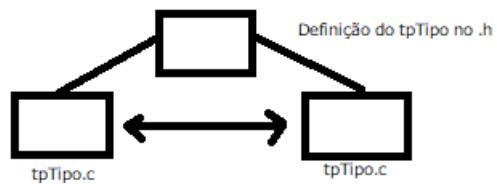


Figura 8:

2.2 Interface em Detalhe

Sintaxe Regra dos dados



Figura 9:

Semântica Significado dos dados

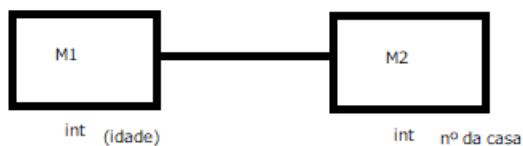


Figura 10:

2.3 Exemplo de Interface

Interface Explicita e Implícita **Explicita**:
tpDadosAluno *obterAluno(int id)

- Interface esperada pelo cliente: ponteiro válido referenciando os dados de aluno ou NULL
- Interface esperada pelo servidor: id válido
- Acesso aos dados de aluno, se o mesmo existir **Interface Implícita**
- Interface esperada por ambos: tpDadosAluno (Interface fornecida por terceiro)

Obs: protocolo de uso: formas de se utilizar os itens que compõem uma interface para que esta possa ser operada corretamente (na documentação de cada função)

2.4 Processo de Desenvolvimento

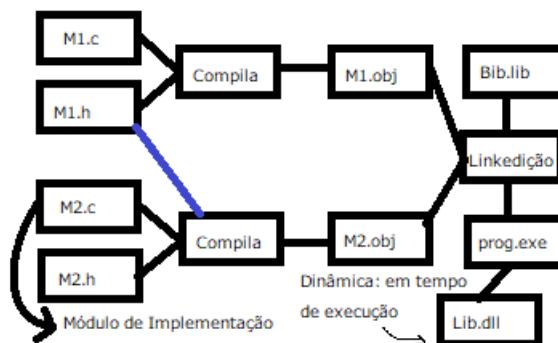


Figura 11:

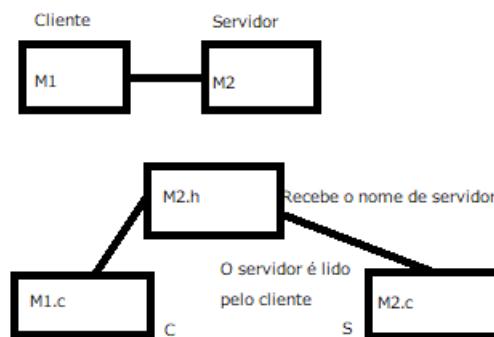


Figura 12:

2.5 Bibliotecas Estáticas e Dinâmicas

2.5.1 Estática

Vantagem: .lib já é acoplado em tempo de linkedição à aplicação executável **Desvantagem:** Existe uma cópia da biblioteca estática na memória para cada executável que a utiliza

2.5.2 Dinâmica:

Vantagem: Só é carregada uma instância em memória, independente do número de aplicações que a utilizam **Desvantagem:** .dll precisa estar na máquina para a aplicação funcionar (dependência externa)

2.6 Módulo de definição (.h)

- Especificação externa voltada para o programadores do módulo cliente
- Protótipos ou assinaturas das funções de acesso
- Declarações e códigos públicos ao módulo

2.7 Módulo de Implementação (.c)

- Especificação interna voltada para os programadores do módulo servidor
- Protótipos ou assinaturas das funções internas
- Declarações e códigos encapsulados no módulo
- códigos executáveis das funções

2.8 Tipo Abstrato de Dados (TAD)

É a estrutura encapsulada que somente é conhecida pelos clientes através da função de acesso.

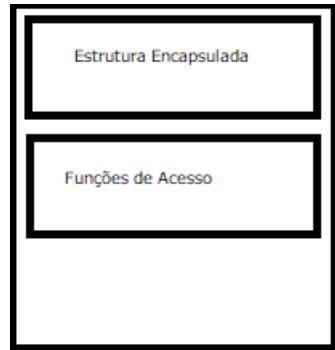


Figura 13:

3 Aula 03 - 20/03/2019

3.1 Conceito

Toda estrutura de dados possui "cabeça".

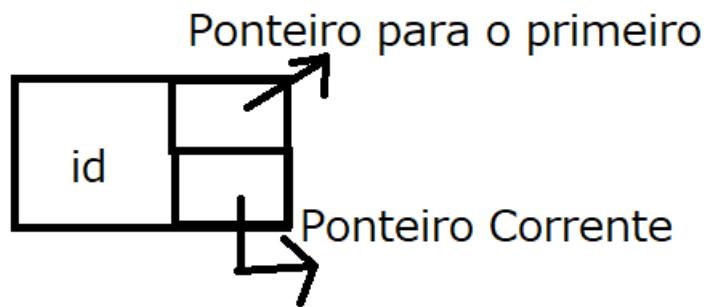


Figura 14:

3.2 Singleton

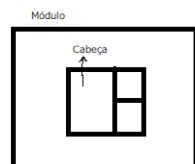


Figura 15:

3.3 Normal

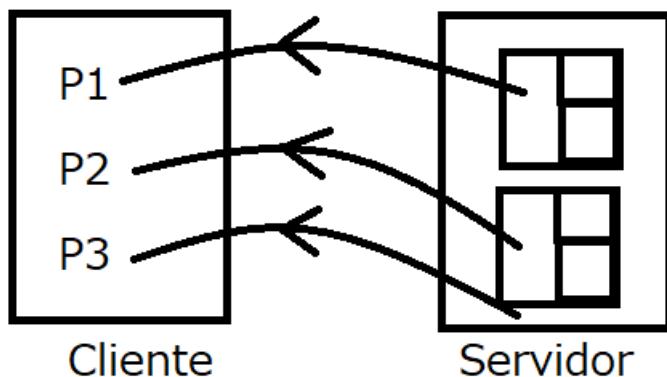


Figura 16:

criaArv() - O ponteiro é passado por referência pois o ponteiro tem que ser passado para fora e pq é uma cópia - Cria uma cabeça
criaLista(ptLista1) ptLista1 = ptColuna
crialista(ptlista2) ptLista2 = ptLinha1
criaLista(ptLista3) ptLista3 = ptLinha2
criaNo (ptLinha1, ptVal1)
criaNo (ptLinha2, ptVal2)
. . .

4 Aula 04 - 25/03/2019

4.1 Propriedade da Modularização

- Encapsulamento - Proteção: propriedade relacionada com a proteção dos dados de um componente de forma que este possa ser utilizado sem perder suas características básicas. Ex: televisão e seus botões
- Acoplamento
- Coesão: Conceito

4.2 Encapsulamento

Vantagem: facilitar a manutenção, pois tudo relacionado à estrutura encapsulada está dentro de um determinado módulo. Facilita a documentação do que se encontra dentro do módulo. **Desvantagem:** módulo específico demais. (do exagero)

Tipos de Encapsulamento

- Código: código de uma função de acesso contida em um módulo de implementação e que não é vista pelo módulo cliente.
- FOR, WHILE, DO WHILE: também é um encapsulamento de código
- Módulo
- Função

Uma aplicação que utiliza ponteiro pode destruir o encapsulamento, pois acessa direto a memória

Variável:

- static: variável global a classe—módulo
- local: encapsulada no bloco de código
- private: encapsulada no objeto
- protected: estrutura de herança

Documentação

- Interna: módulo de implementação/Servidor (.c)
- Externa: Módulo de interface/Cliente (.h)
- De uso: usuário

4.3 Acoplamento

Propriedade relacionada com a interface entre os módulos

1. Interface
2. Conector: é o item da interface. Ex: protótipo da função, arquivo, variável global

3. critério de qualidade do acoplamento: É o mais simples/menor possível. (Tamanho de conector). Quantidade de pârâmetros de uma função. Solução: agrupar parâmetros em structs
4. Quantidade de conector: Todas as funções internar e externas no (.h). Tem que saber quais colocar no (.h). Verificar se todos os itens são necessários e suficientes.
5. Complexidade do Conector: Protocolo de uso. (Se a documentação é bem feita, o conector difícil pode ser facilmente utilizado)

4.4 Coesão

Propriedade relacionada com o grau de interdependência dos elementos que compõem o módulo (conceito). Melhor coesão: um único conceito estabelecido **Níveis de Coesão**

1. Incidental: bagunça. Não há relação entre os vários conceitos inseridos no módulo
2. Lógica: Agrupar logicamente os conceitos. Os elementos possuem uma relação lógica entre os conceitos de uma forma possivelmente genéricas. Ex: módulo que calcula tudo.
3. Temporal: Os elementos estão relacionados pela necessidade de serem utilizados no mesmo período de tempo
4. Procedural: Os elementos estão relacionados pela necessidade de serem utilizados na mesma ordem (.bat)
5. Funcional: Agrupando por funcionalidade. Os elementos estão relacionados por funcionalidade. Tudo relacionado à mesma funcionalidade está no mesmo módulo.
6. Abstração de Dados: Um único conceito

5 Aula 05 - 27/03/2019

6 Aula 06 - 01/04/2019

7 Aula 07 - 03/04/2019

Aula copiada em um caderno em anexo! Foi Copiada no Tablet

03/04/19 - Quarta-feira

Requisitos

Especificação de Requisitos

1) Requisitos

- ↳ O que deve ser feito
- ↳ Nunca como deve ser feito

2) Características dos Requisitos

- ↳ claros e diretos
- ↳ Linguagem natural

3) Etapas da especificação

- ↳ elicitação (conversa com o cliente)

1) Entrevista

2) Brainstorm

3) Questionário

↳ documentações

1) Requisitos genéricos e específicos

2) Contrato

↳ verificação com a equipe

1) análise para ver se dá para implementar

↳ se a documentação somente possui requisitos computáveis

2) Juntos com a equipe técnica.

↳ validação

1) cliente (assinatura do contrato)

4) Tipos de Requisito

↳ funcional

1) Regras que devem ser implementadas na especificação relacionadas com o negócio.

↳ não funcionais

1) propriedades que a aplicação deve possuir e que não necessariamente estão relacionadas com o negócio.

ex: login e senha → requisito de segurança

a) Segurança (login / senha)

b) Disponibilidade (24x7)

c) Backup

Requisitos

03/04/2019

- d) velocidade (Todas as consultas devem retornar resultado no máximo 3s)
 ↳ inverso
 ↳ O que se compromete a não fazer.

5) Exemplos de Requisitos

a) Bem formulados

- para cada aluno deve ser cadastrado matrícula e nome
- o relatório de turmas deve ser disponibilizado no 1º dia de matrícula.

b) Mal formulados

- a interface deve ser de fácil utilização
- o relatório apresenta seus dados mais recentários

08/04/2019

Segunda

Assertivas estruturais

Lista

$pCom \rightarrow pAnt \neq NULL$

$pCorr \rightarrow pAnt \rightarrow pProx == pCom$.

Se $pCom \rightarrow pProx \neq NULL$

$pCom \rightarrow pProx \rightarrow pAnt == pCom$

Árvore

- ponteiro de um nó de árvore a esquerda.

nunca aponta para o par num para nó de sub-árvore a direita.

Obs:ontagem do Trabalho

- Modelo
- Assutivas
- Exemplo

→ São funções, condições, atributos, propriedades ou características a serem satisfeitas pelo artifício.

Estudos para o trabalho 2

10.1 O que são Requisitos

10. Especificação de Módulos

↳ Apêndice 2

Padrões para especificações

- ↳ Funcionais, não-funcionais, universo, restrições
- ↳ funcionalidade ↳ desempenho, ↳ ex: "o módulo será redigido em C".
- ↳ desejada para um software facilidade de uso

→ ex: o módulo fará isso

Requisitos Funcionais

↳ Especificam as funções que o artifício deve ser capaz de executar; especificam o serviço a ser prestado pelo artifício.

Requisitos Não-funcionais

↳ propriedades que o artifício deve possuir
ex: segurança → não oferece riscos perigosos, materiais, ecológicos ou financeiros além dos toleráveis, tendo em vista a natureza do serviço.

Padrões para a especificação de Requisitos

1) Análise de domínio

↳ Separa o problema a resolver em duas partes. Uma que é genérica e outra que é específica.

A parte genérica consiste na parte comum a outros softwares.
ex: Em um software para uma escola, é comum manter dados dos alunos...

A parte específica resolve o problema específico do cliente, por exemplo, um cliente pode solicitar um sistema que mida a entrada e da saída do aluno.

Na parte genérica, o software permanece quase inalterável, enquanto na parte específica ele sofre atualizações.

Pag 34] → Exemplo de especificação de uma função.

Pag 362 → Assiniva de entrada e saída de funções

10.1.5 Processo de identificação e requisição

2) A identificação de Requisitos em Si

01. Identificam o escopo do artefato

↳ são definidos os objetivos, a abrangência, os potenciais usuários envolvidos, e são esboçados os serviços a serem prestados.

02. Identificam Requisitos

↳ São elicitados os requisitos através da interação com pessoas

7.1 Requisitos

- 7.1.1 O que são Requisitos
- 7.1.2 Característica dos Requisitos
- 7.1.3 Etapas da Especificação
- 7.1.4 Tipos de Requisitos
- 7.1.5 Exemplos de Requisitos
- 7.1.6 Vetor

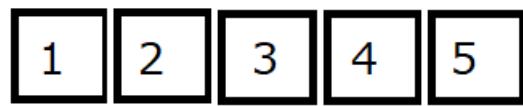


Figura 17:

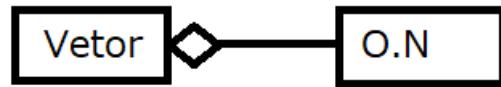


Figura 18:

- 7.1.7 Lista Simplesmente Encadeada Com Cabeça

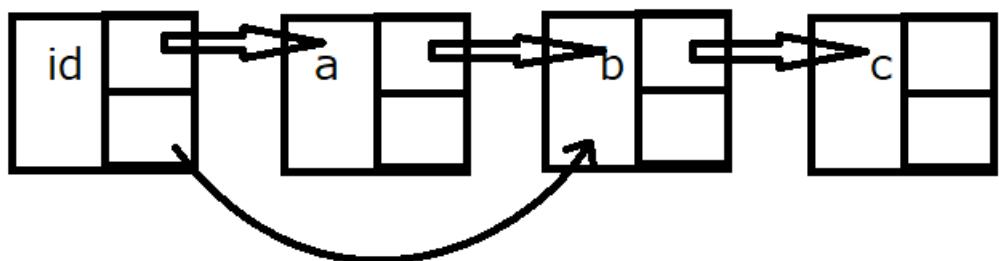


Figura 19:

7.1.8 Lista Duplamente Encadeada Com Cabeça

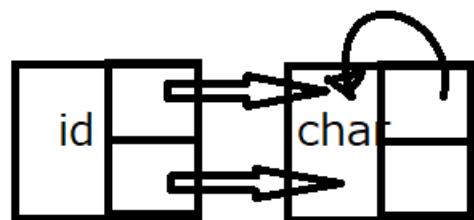


Figura 20:

7.1.9 Arvore Binaria Generica Com Cabeça

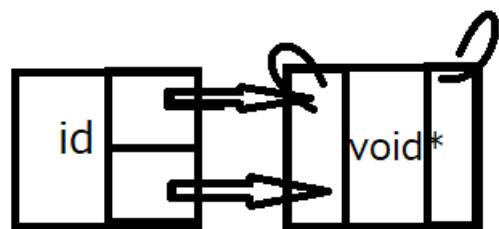


Figura 21:

8 Aula 08 - 08/04/2019

Aula copiada em um caderno em anexo! Foi copiada no tablet

8.1 Assertivas Estruturais

9 Aula 09 - 15/04/2019

9.1 Vetor de Lista de arvores

Esse desenho está em um caderno anexo! Copiado na mão

10 Aula 10 - 17/04/2019

10.1 Assertivas

Regras consideradas validas ao executar um determinado ponto do programa São utilizadas em:

- argumentação de corretude
- instrumentação (codigo externo para verificar a corretude)

10.1.1 Assertivas Estruturais

São regras que complementam o modelo de uma estrutura de dados

10.1.2 Assertivas de Entrada e Saída

Assertiva de entrada + Bloco de comando = Assertiva de saída

OBS: As assertivas precisam estar corretas e completas **Exemplo:**

AE: Excluir o no corrente intermediario de uma lista duplamente encadeada com cabeça

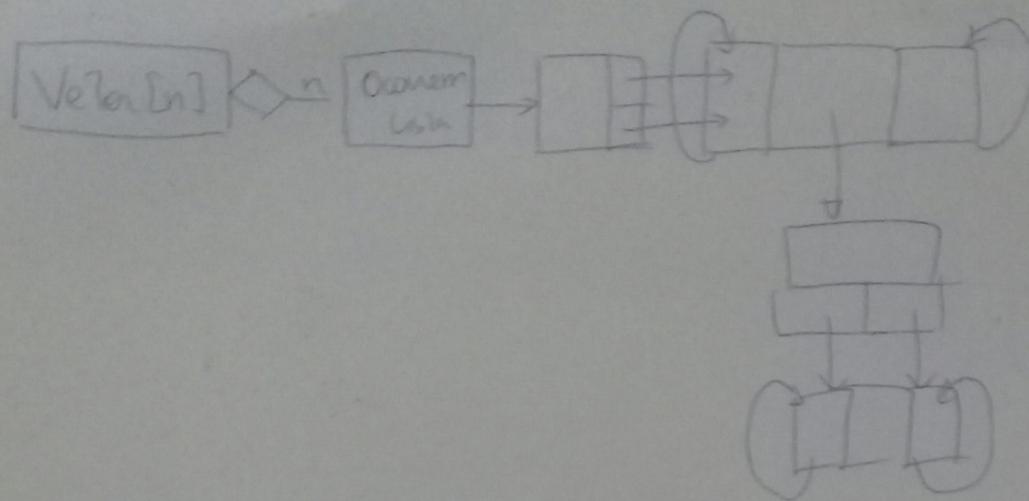
AS:

AE: Lista existe e possui pelo menos 3 nós. Ponteiro corrente aponta para o nó intermediario que quer excluir. Valem as assertivas estruturais da lista duplamente encadeada com cabeça

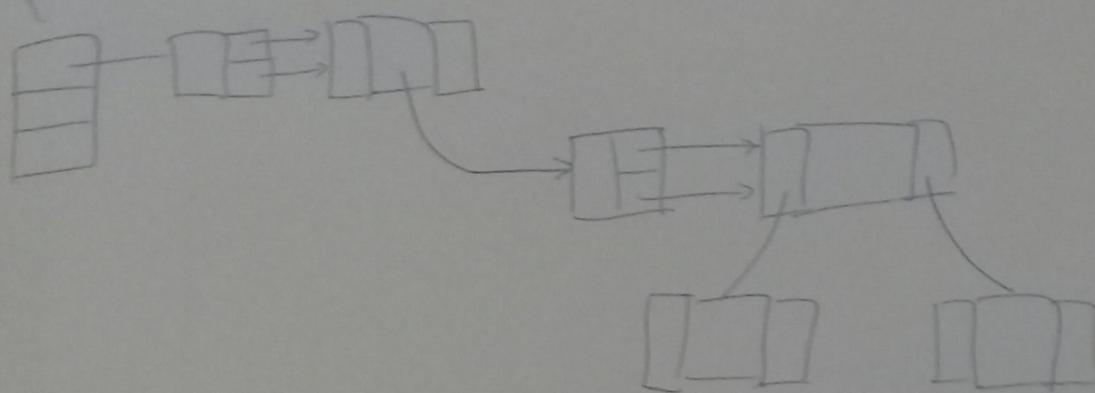
AS: O nó foi excluido. Valem as assertivas estruturais da lista duplamente encadeada com cabeça. Ponteiro corrente aponta para o primeiro nó da lista.

• Vetor Lista de Arvores

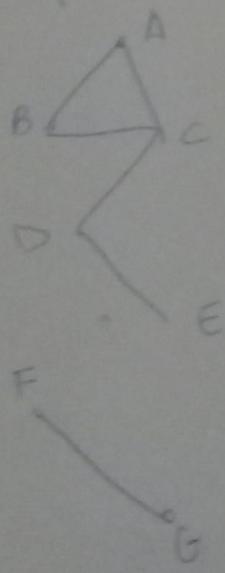
15/09/2019



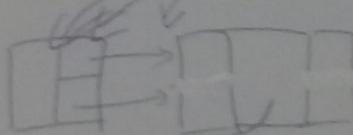
Exemplo



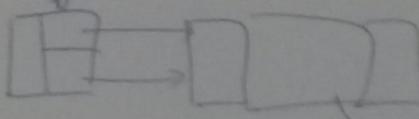
Grafo



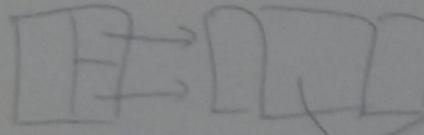
Vertice



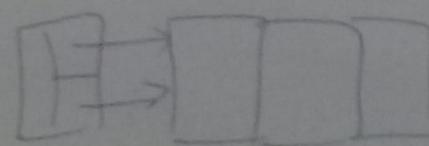
Aresta



Vertices



Oligos



11 Aula 11 - 22/04/2019

12 Aula 12 - 24/04/2019

Aula copiada no tablet!

12.1 Implementação da Programação Modular

12.1.1 Espaço de Dados

12.2 Tipo de dados

12.3 Tipos Básicos

13 Aula 13 - 29/04/2019

Essa aula está no caderno em anexo! Foi copiada no tablet

13.1 Pré-processamento

13.2 Exercícios da Lista

Foi dado da questão 12 até a 14

14 Aula 14 - 06/05/2019

Essa aula está no caderno em anexo! Foi copiada no tablet

KEY POINTS

Implementação de Programação Modular

Espaço de Dados

Tipos de Dados

NAME/DATE/SUBJECT

24/04/2019

NOTES

Implementação da Programação Modular

1) Espaço de dados

São áreas de armazenamento:

- Possui um tamanho
- Possui um ou mais nomes de referência
- Alocados em um meio

exemplos

$A[j]$ → j-ésimo elemento do vetor A

* ptAux → espaço apontado por ptAux

ptAux → espaço que contém um endereço

$\rightarrow \text{ptElemTabSimb}^*$ $\text{ObterElemTabSimb}(\text{char}^* \text{ptSimbolo})$

(+ $\text{ObterElemTabSimb}(\text{char}^* \text{ptSimbolo})$). Id

proto tipo
da função

de

espacos
de dados

$\rightarrow \text{ObterElemTabSimb}(\text{char}^* \text{ptSimbolo}) \rightarrow \text{Id.}$

sub campo id do elemento retornado pela função acima)

2) Tipos de Dados

Determinam:

→ organização

→ codificação

→ tamanho em bytes

→ conjunto de valores permitidos.

↳ ENUM

como um binário é interpretado

XX XXXX, dígito do CPF, matrícula

int < float

SUMMARY

KEY POINTS

Espaço de Dados

NAME/DATE/SUBJECT

24/04/2019

NOTES

obs 1:

Um espaço de dados precisa estar associado a um tipo para que possa ser interpretado por um programa desenvolvido em uma linguagem tipada

↳ não permite a criação de variáveis sem tipo.

obs 2: Tipos de tipos

Tipos computacionais

int, char, short, float

Tipos Básicos

struct, union, enum, typedef

Tipos Abstratos de dados

estruturas de dados encapsuladas ...

3) Tipos Básicos

- struct



mexer em c1 não afeta c2

- Union



1 int c1

float c2

chan c3 4

type cache

SUMMARY

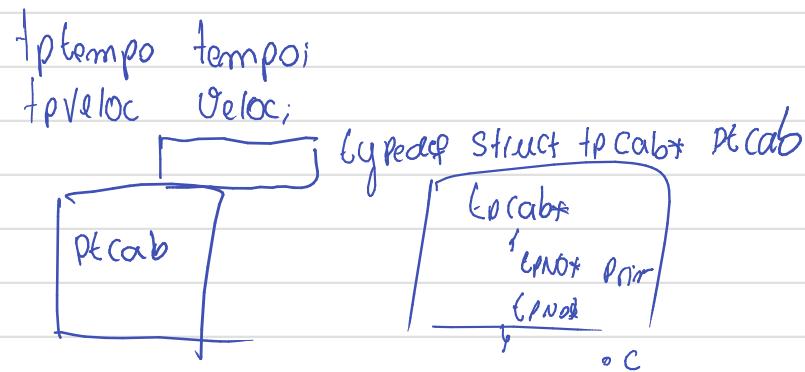
KEY POINTS

NAME/DATE/SUBJECT

NOTES

Enum
{ a → 0
 b → 1
 c → 2
 d,
 e
 ↳ - type def

typedef float tpVeloc;
typedef float tpTempo



SUMMARY

NOTES

4) Declaração e definição de elementos

definir: aloca espaço de dados e amarra o espaço ao nome (binding)

declarar: corresponde o espaço a valores de um determinado tipo

$\rightarrow \text{int } x;$ → definir e declarar

malloc → define

tipo → declarar
struct

5) Implementação e C e C++

a) declarações e definições de nomes globais exportados pelo módulo Servidor

$\rightarrow \text{int } a;$

b) declarações externas contidas no módulo Cliente e que somente declaram o nome sem associá-lo a um espaço de dados.

$\rightarrow \text{extern } \text{int } a;$

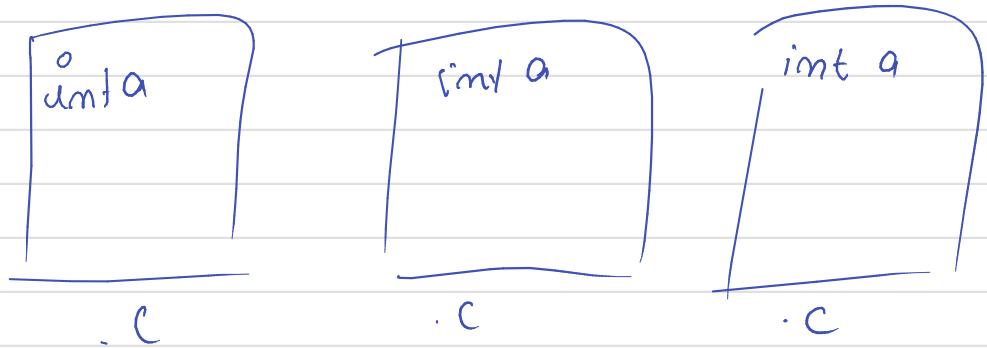
KEY POINTS

NAME/DATE/SUBJECT

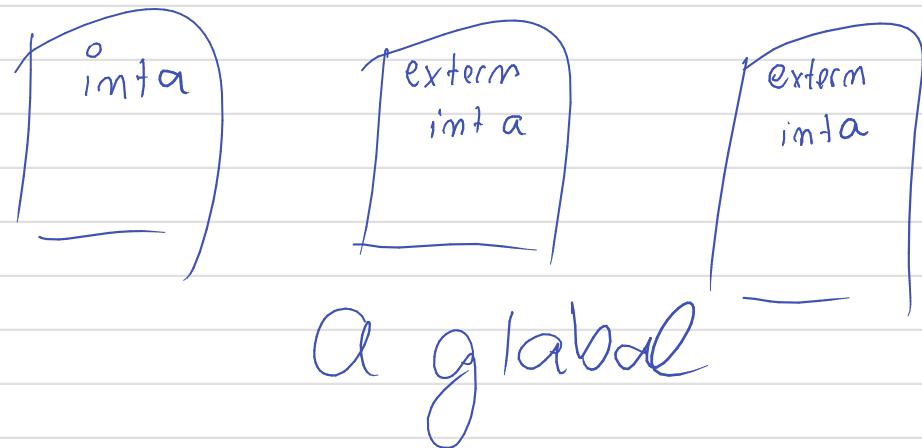
NOTES

c) Declarações e definições de nomes globais encapsulados no módulo

Static int a;



a é global



a global

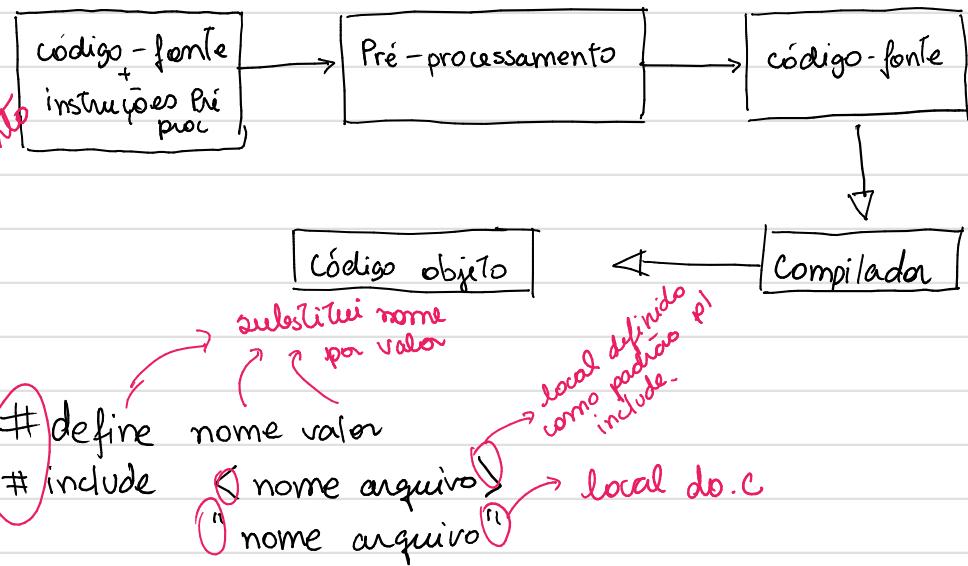
SUMMARY

29/04/2019

NOTES

6) Pré-processamento

instruções de pré-processamento
não são comandos em C



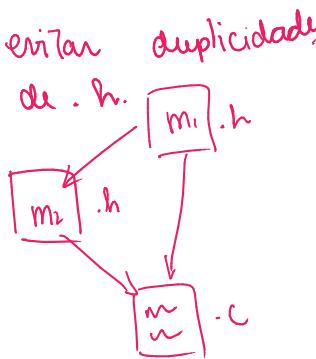
#if defined (nome) ou #ifdef nome
textos

#else
textos
#endif

#if !defined
ou
#ifndef

#if !defined (Exemp_mod)
#define Exemp_mod
corpo do .h
#endif

(*)
Exemp EXT int versão[?]
#if defined (Exempla_own)
= { 1, 2, 3, 4, 5, 6, 7 } ;
#else
;
#endif



SUMMARY

④ #ifdef exempl_own
#define exempl_ext
#else
#define exempl_ext extern
#endif

NOTES

```
#define exempl_own  
#include "m1.h"  
#undef exempl_own  
  
#include "m2.h"
```

Exercícios da lista

- 12) Apresente uma situação de definição sem declarar a variável.
- 13) É possível considerar que apenas declarar sem definir, chega a definir um espaço de dados? (Centrado / Talvez) Justifique sua resposta.
- 14) Como é possível personalizar interfaces para módulos cliente sem duplicar códigos? Apresente exemplo.

06/05/2019

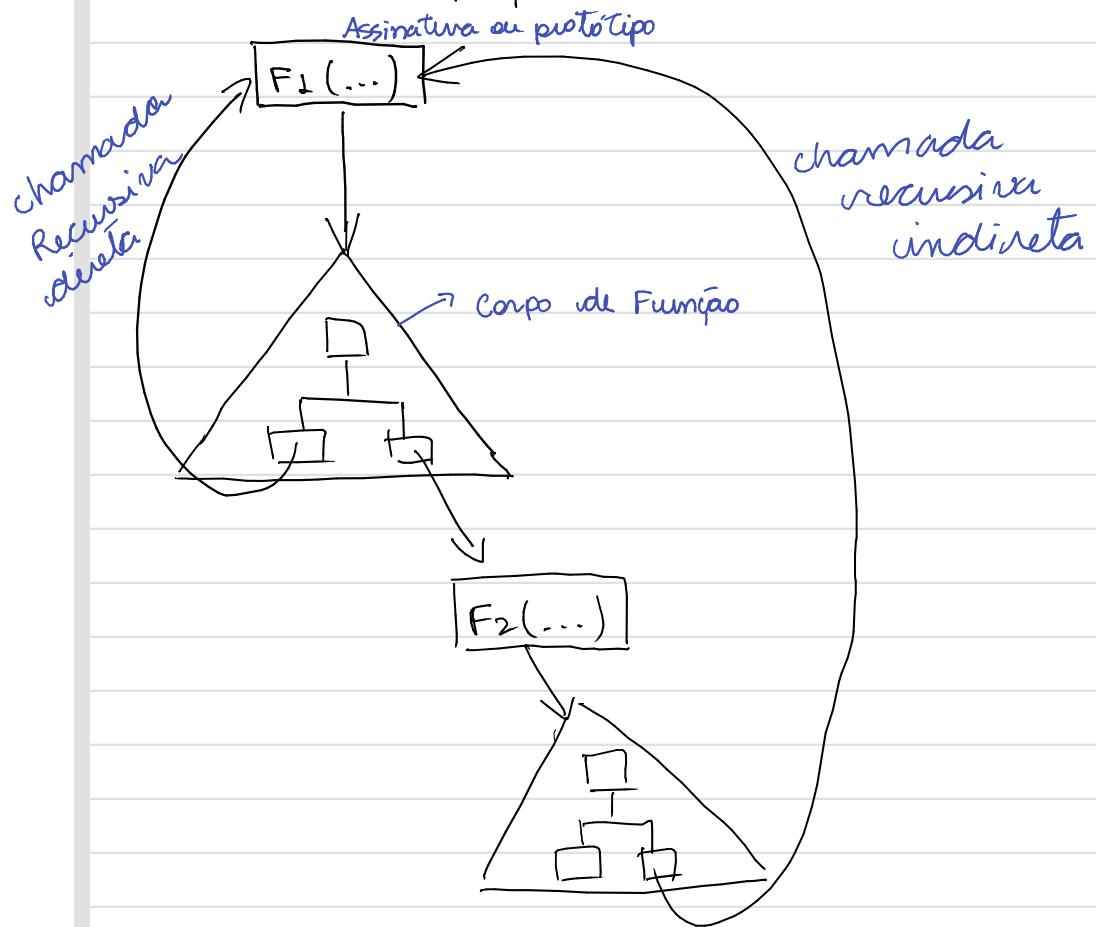
Estrutura de Funções

- 1) Paradigma
 - Forma de programar
 - Procedural
 - Orientada a Objetos
 - POO
 - Programação Modular
- Receita de Bolo.*

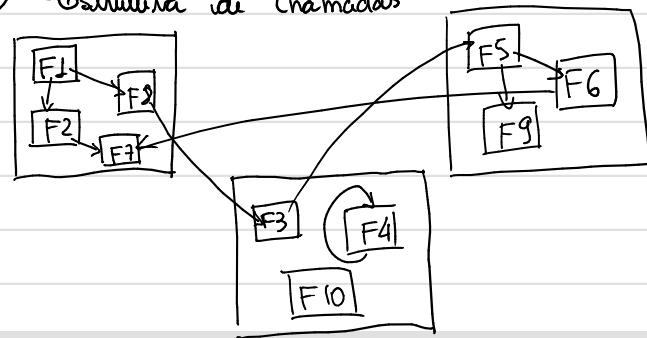
05/06/2019

NOTES

2) Estrutura de funções



3) Estrutura de chamadas



SUMMARY

06/05/2019

NOTES

Acos de chamada

$F_4 \rightarrow F_4$ chamada recursiva direta

$F_9 \rightarrow F_8 \rightarrow F_3 \rightarrow F_5 \rightarrow F_9$ chamada recursiva indireta

F_{10} função morta

$F_8 \rightarrow F_3 \rightarrow F_5 \rightarrow F_6 \rightarrow F_7$ dependência circular entre módulos

↳ não é recursiva pq não começa e termina no mesmo lugar.

F1 Origem

4) Função

é uma porção autocontida de códigos. Possui um nome, uma assinatura e um ou mais corpos de código. (ponteiro para função)

5) Especificação de Função } no .h

- Objetivo

- Se o nome é auto explicativo esse item pode ser retirado.

- Acoplamento

- Parâmetros e condições de retorno

- Condições de Acoplamento

- Assentiva de entrada e saída.

NOTES

- Interface com o usuário
 - mensagem, mostrar informações do tabuleiro ...
- Requisitos
 - Tópicos dizendo o que a função faz
- Hipóteses
 - Regra que considera válida antes do desempenhamento da função
- Restrições
 - "não pode entregar depois de turça"
 - Regras que restringem as alternativas de soluções utilizadas no desenvolvimento de uma aplicação.

c) Housekeeping

↳ são funções em cada módulo.

Módulo de bloco de código responsável por liberar recursos alocaados a programas, componentes ou funções ao terminar a execução.

↳ módulo lista fib?

Assentivas → no doc ou nos comentários?

↳ e em funções que não tem parâmetro?
Todo ponto.

14.1 Paradigma

14.2 Estrutura de Funções

14.3 Estrutura de Chamadas

14.4 Função

14.5 Especificação de Função

14.6 House Keeping

15 Aula 15 - 08/05/2019

15.1 Repetições

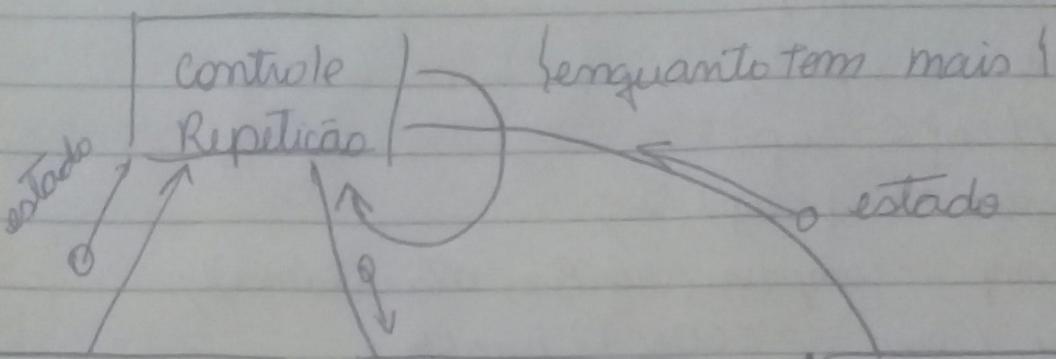
15.2 Recursão

15.3 Estado

15.4 Esquema de Algoritmo

15.5 Parâmetros do tipo ponteiro para função

7) Repetições

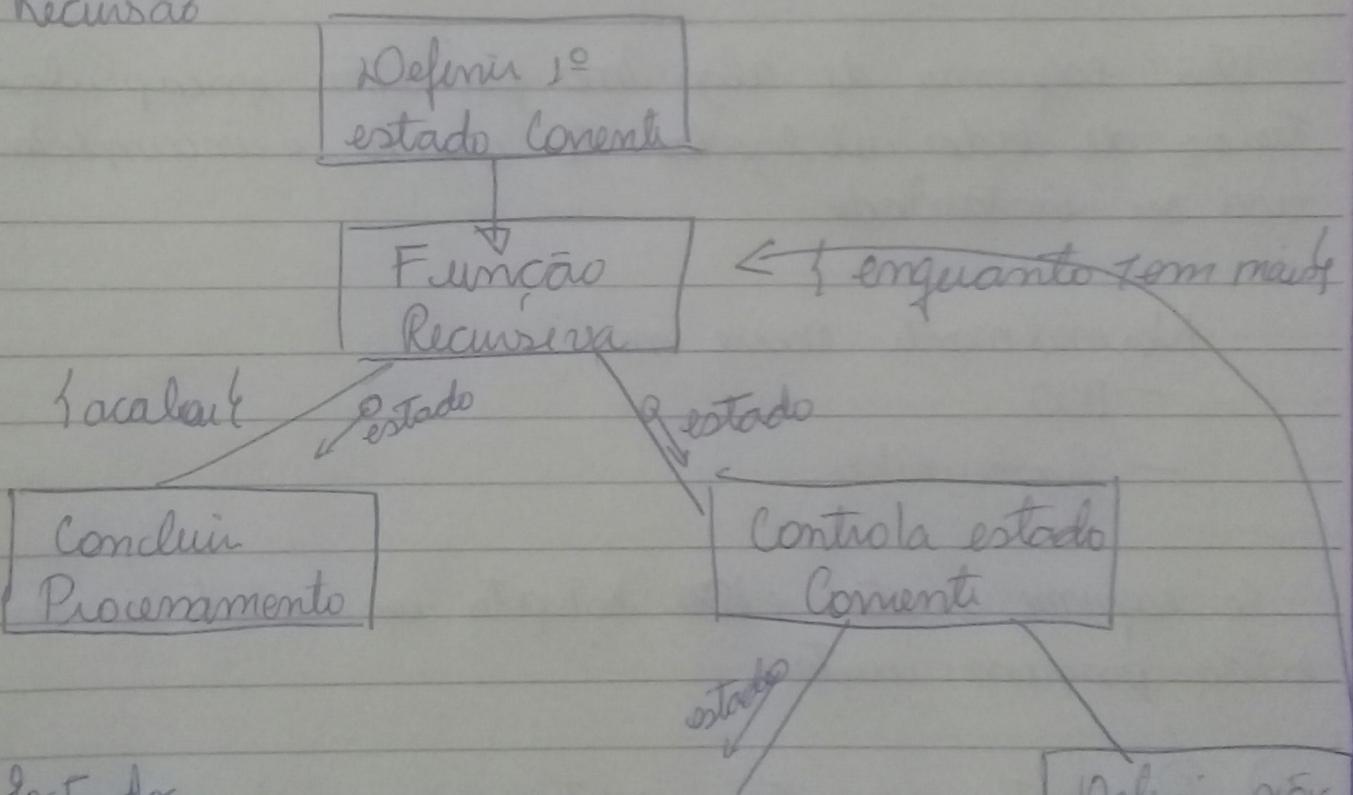


Definir o 1º
estado Converte

Procurar
1º estado Converte

Definição
mais 1º
estado Converte

8) Recursão



↓
↓
↓

↓
↓
↓

↓
↓
↓

9) Estado

Descriptor de estado:

- variável ou variáveis que
definem um estado

ex: busca sequencial \rightarrow int i;

busca binária \rightarrow int inf; int sup;

estado: valoração de descriptor de estado.

Procurar estado
Converte

Definição
mais 1º
estado Converte

10) Esquema de Algoritmo

```
inf = ObterLimInf();
sup = ObterLimSup();
while(inf <= sup) {
    meio = (inf + sup) / 2;
    comp = comparar(valorBuscado, obterValor(meio));
    if (comp == IGUAL) { break; }
    if (comp == MENOR) { sup = meio - 1; }
    else { inf = meio + 1; }
```

4

(Obs: esquema de algoritmo permitem encapsular estruturas de dados utilizadas. É curto, é incompleto e precisa ser instanciado.

Normalmente ocorre em:

- POO
- frameworks

Se esquema curto e ~~é~~ hotspots com assertivas validadas dentro próprio curto.

11) Parâmetros do tipo ponteiro para funções

```
float areaQuad(float base, float altura)
{ return base * altura; }
```

```
float areaTri(float base, float altura)
{ return base * altura / 2; }
```

S T Q Q S S D

08 / 15 / 2017

int procenaArea (float val1, float val2, float l² func (float
float))

}

printf ("%f", func (val1, val2));

4

:

CondRit = procenaArea (5, 2, areaQuad);

condRit = procenaArea (3, 2, areaTri);