

ModularIntrodução

↳ Vantagens da programação modular:

- Vencer barreiras de complexidade
- Distribuir tarefas em grupo
- Reuso de módulos (reduz esforço futuro)
- Criação de um acervo de módulos reutilizáveis
- Permite a criação e trabalho com base-lines de módulos já testados
- Desenvolvimento incremental
- Aprimoramento individual
- Reduz tempo de compilação.

↳ Princípios de modularidade:

1. Módulo:

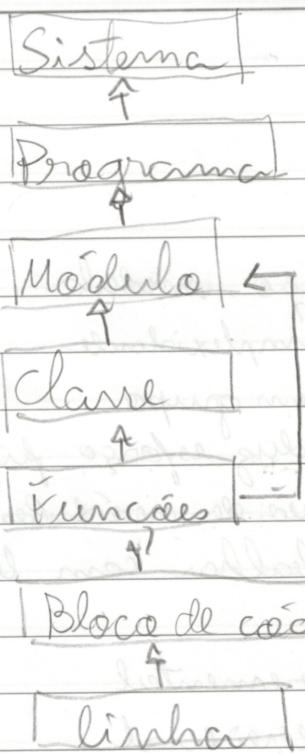
a) Definição física: Unidade de compilação independente.

b) Definição lógica: Pode ser um único algoritmo

2. Hierarquia:

(Podem ser: Bloco de códigos, fragmentos de texto de documentações, funções, figuras ou diagramas, sessões de documentação, tipos de dados, classe, componentes...)





- Artefato: algo elaborado durante o processo de desenvolvimento e que possua identidade própria. Consegue ser reutilizado
- Construto: Versão de aplicação, executável, podendo ser incompleta

### 3. Interface

- Mecanismo de troca de dados, comandos e eventos entre elementos de programa
- \* Interface sempre ocorre entre elementos de mesmo nível na hierarquia

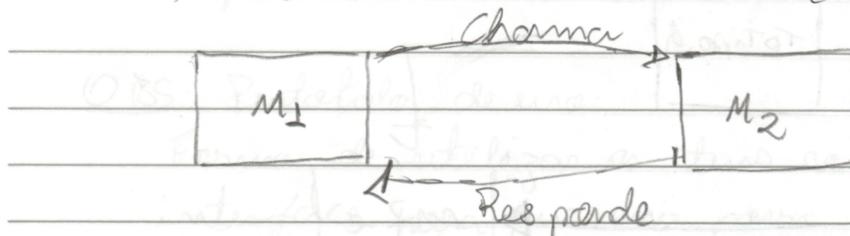
A) Exemplos de formas de interface:

- Sistema + Sistema  $\rightarrow$  Arguidas

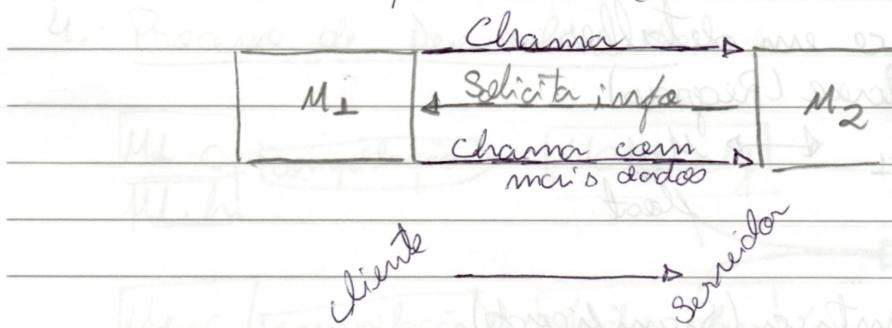
- Módulo + Módulo  $\rightarrow$  funções de acesso e seus parâmetros

- Bloco de código + Bloco de código  $\rightarrow$  Variaíveis globais

B) Relacionamento Cliente-Servidor:



- Caso especial: callback



Ocorre uma inversão temporária na relação cliente-servidor, uma vez que o servidor requer mais informações que precisa do cliente.

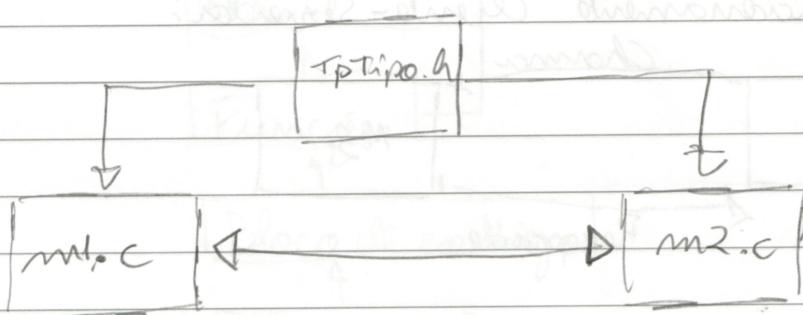
c) Interface fornecida por parceiros:

~~E quando um tipo é licenciado~~  
~~interface entre 2 módulos não está~~  
~~definida~~

1 /

### c) Interface fornecida por terceiros

É quando um tipo utilizado em uma interface entre dois módulos não está definido em nenhum dos módulos de implementação e sim em um módulo de definição comum aos dois.



- Interface em detalhes:

- sintaxe (Regras)

$$M_1 \leftrightarrow M_2$$

int      float

- Semântica (Significado)

$$M_1 \rightarrow M_2$$

idade      meu casa  
(int)      (int)

### Exemplos de interface:

Tp Dados Aluno \* obterAluno (int id)

(Assinatura / protótipo) (Interface explícita)

- Interface esperada pelo cliente:

Ponteiro nô nulo referenciando os dados do aluno ou NULL

- Interface esperada pelo servidor:  
id válido

Acesso aos dados do aluno no mesmo contexto  
(interface implícita)

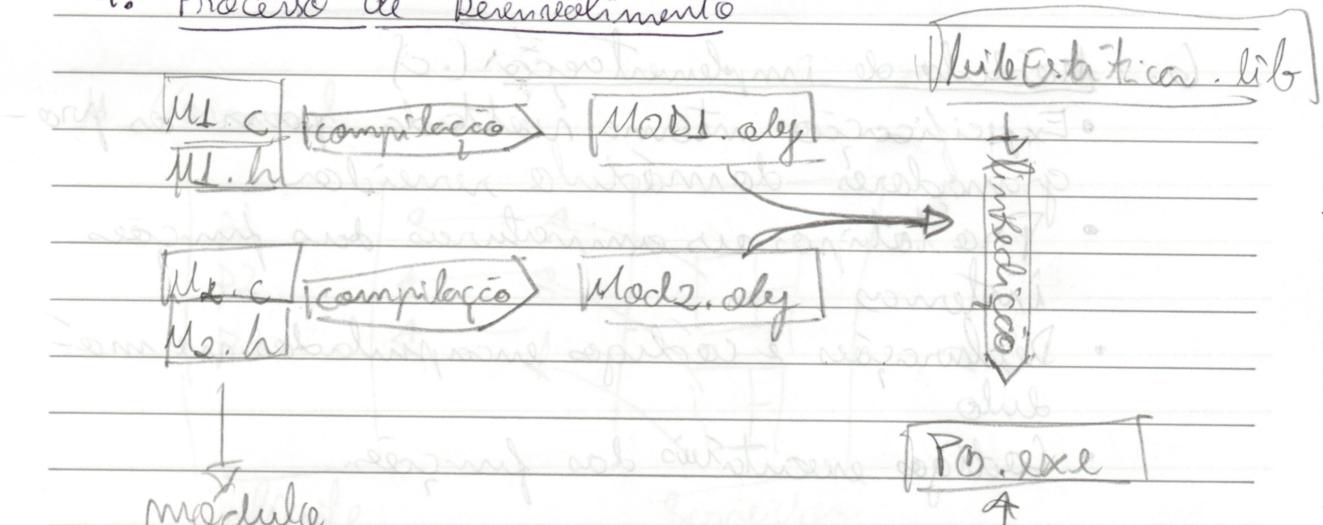
- Interface esperada por clientes:

tipo DadosAlunos (interface fornecida por terceiros)

OBS: Protocolo de uso:

Fórmula de utilizar os items compõem uma interface para que essa possa ser operada corretamente

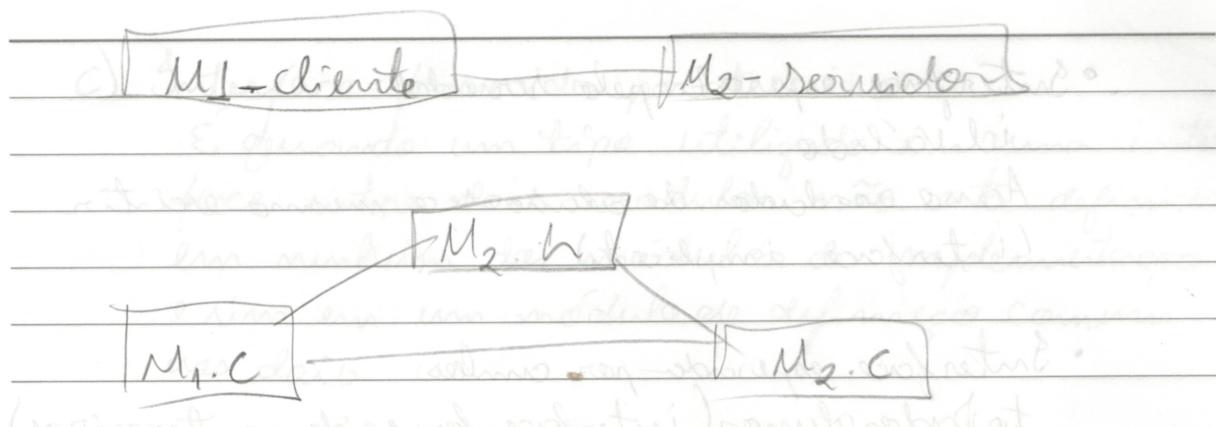
#### 4. Processo de Desenvolvimento



modulo

de definição

1 /



### 5. Módulo de definição (.h)

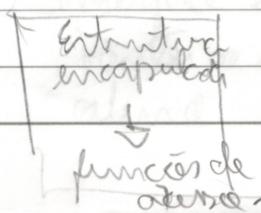
- Especificação externa redigida para os programadores do módulo cliente
- Protótipos (encapsulados) das funções de acesso
- Declarações e códigos públicos do módulo

### 6. Módulo de implementação (.c)

- Especificação interna redigida para os programadores do módulo servidor
- Protótipos das anotações das funções internas
- Declarações e códigos encapsulados no módulo
- Códigos executáveis das funções.

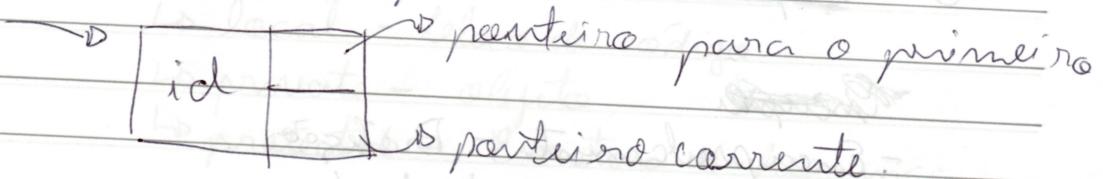
### 7. Tipos abstratos de dados (TAD)

- É uma estrutura encapsulada que só mente é conhecida pelos clientes pelas funções de acesso.

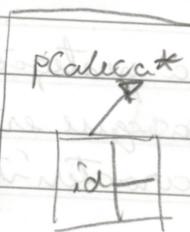


11

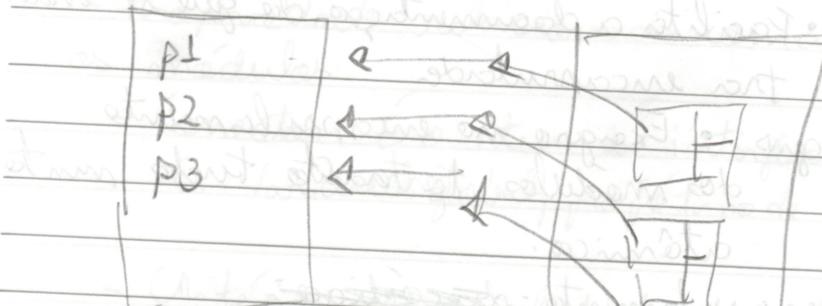
Conceito: Toda estrutura ~~em�ntata~~ de dados tem "cabeça".



Singletão



Normalmente os elementos



cliente

Serializar

~~(8)~~

## 8. Propriedades da modularização

~~Propriedades~~

- Encapsulamento - Proteção.
- Acoplamento - interface
- Coesão - conceito

### 9. Encapsulamento

Propriedade relacionada com a proteção das dados de um componente de forma que este possa ser utilizado sem perder suas características técnicas.

Vantagens:

- Manutenção facilitada, pois tudo relacionando à estrutura encapsulada está dentro do mesmo módulo.
- Facilita a documentação de que se encontra encapsulada.

Desvantagens:

- Exigir no encapsulamento dos módulos, tanto que tudo é muito atônico.

- Tipos de encapsulamento: ~~de código~~:

• De código

• Código da função de acesso contida no módulo de implementação, não visto pelos módulos clientes

• Blocos de repetição (for, while, etc)

• Código <sup>internos</sup> de funções.

• Blocos de decisão (if, entre...).

• De variáveis, ~~static~~

↳ static - classe ou módulo

↳ local - local de código

↳ private - objeto

↳ protected - Estrutura de herança

• De documentação:

↳ Interna, voltada para os programadores do servidor (.c)

↳ Externa, voltada para os programadores do cliente (.h)

↳ De uso, voltada para o usuário

## 10. Acoplamento

Propriedade relacionada com a interface entre os módulos

- Conector: É o item da interface

Ex: Variável global, no tipo de função, arquivos,

- Critérios de qualidade de acoplamento

① Tamanho do conector (gt menor, melhor)

↳ Ex: função com 10 parâmetros. função com 2 parâm.

② Quantidade de conectores (gt menor, melhor)

↳ Ex: Interface carregada com muitas funções

↳ Ex: interface errada e fácil de usar

③ Complexidade do conectar: (Boa documentação)

Ter uma boa documentação ou protocolo de uso

## II. Coesão

Propriedade relacionada com o grau de interdependência dos elementos que compõe o módulo (conceito)

- Níveis de coesão:

① Incidental: Não há relação entre as regras concutas que existem no módulo

② Lógica: Os elementos possuem uma relação lógica entre os concutas de uma forma genérica, ex (calcular. lib).

③ Temporal: Os elementos estão relacionados pela necessidade de serem utilizadas dentro do mesmo período de tempo.

④ Procedural: Os elementos estão relacionados pela necessidade de serem utilizados em uma ordem de execução específica

⑤ Funcional: Os elementos estão relacionados por funcionalidades comuns  
(Todos os elementos de uma mesma funcionalidade estão no mesmo módulo)

⑥ Alteração de todos: Um único conceito

## Especificação de Requisitos

### ① Requisitos

- O que deve ser feito. (Definição de MACROS)
- NUNCA como deve ser feito

### ② Características dos requisitos.

- curtos e diretos

- Linguagem natural

### ③ Etapas das especificações

#### - Elicitação

##### TD Técnicas:

- \* Entrevista (Reunião com o cliente, ~~para~~ que lida com o cliente)
- \* Brain storm
- \* Questionário (Disponibilização de um questionário para o cliente)

#### - Documentação

\* Requisitos gerais (Escopo da aplicação)

\* Requisitos específicos (Regras de funcionamento da programação)  
DEVE SER BEM CLARO

#### - Verificação (Reunião com a equipe)

\* Análise da documentação somente para requisitos computacionais.

\* Junto com a equipe técnica

#### - Validação. { validação final }

\* Cliente

1 / 1

## ① Tipos de requisitos

### - Funcional:

Regras que devem ser implementadas na aplicação relacionadas com o negócio

### - Não-funcionais:

Propriedades que a aplicação deve permitir e que não necessariamente estão relacionadas com o negócio

Ex:

↳ Segurança

- login e senha

↳ Disponibilidade

- Funcionamento 24/7.

↳ Backup:

↳ Velocidade

- "Todas as consultas devem retornar resultado no máximo 3 segundos"

### - Imersão

O que a equipe ~~não~~ se compromete a não fazer

## ⑤ Exemplos de requisitos

### a) Bem formulados

~~Para cada aluno deve ser calculado~~

- "Para cada aluno deve ser calculado 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 os dados mais necessárias"

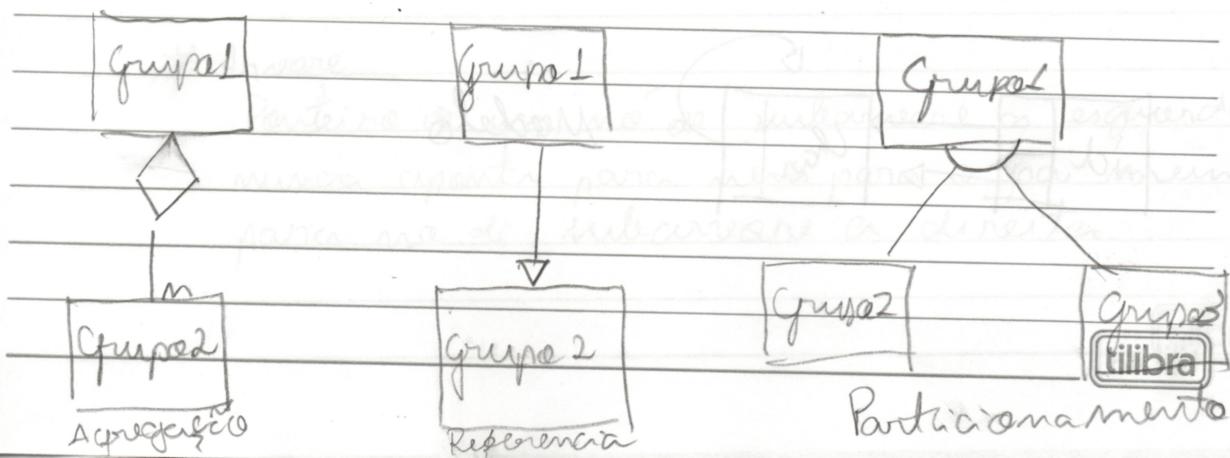
## Modelagem de dados

1 modelo  $\rightarrow$  n exemplos

um modelo deve representar n exemplos

Deve representar somente a estrutura que se comprehende

Notação: UML - lixe



1 / 1

\* Agregação: Grupo 1 compõe por m elementos do Grupo 2

\* Referência: Grupo 1 referencia Grupo 2 (ponteiros)

Exemplos:

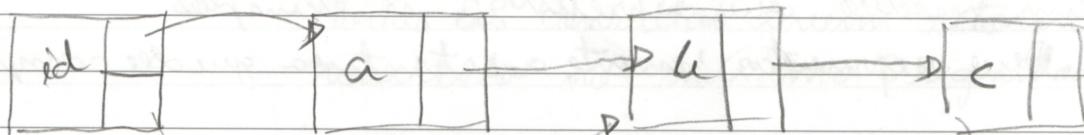
\* Vectors:

1	2	3	5	6
---	---	---	---	---

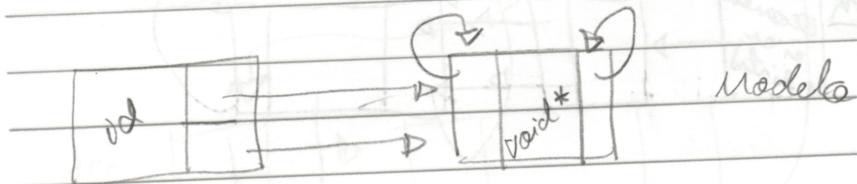
 Exemplo

Vetor  $\rightarrow$  m [ocorrência (múltipla)]

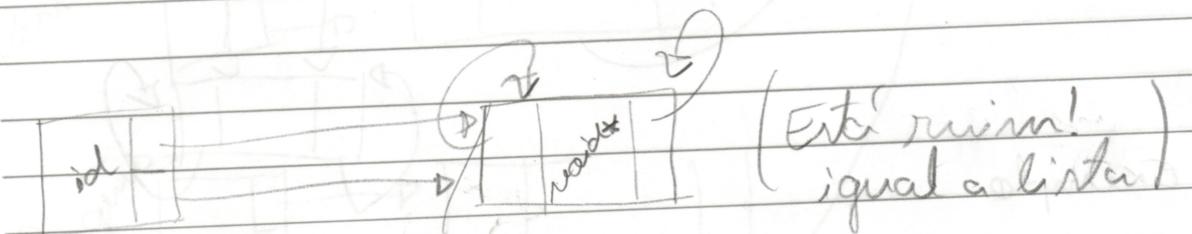
\* lista simplesmente encadeada com caixaçaria



\* lista duplamente encadeada genérica com cabeça



\* Árvore binária genérica com cabeça



Como converter?

↳ Assertivas estruturais

\* lista:

Se  $p_{carr} \rightarrow p_{antes} \neq NULL$

$p_{carr} \rightarrow p_{antes} \rightarrow p_{prox} = p_{carr}$

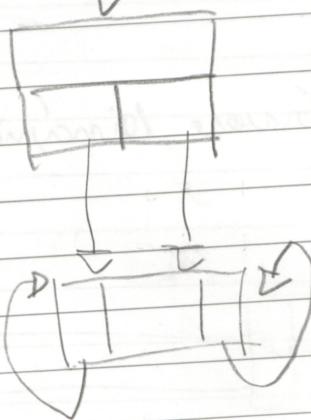
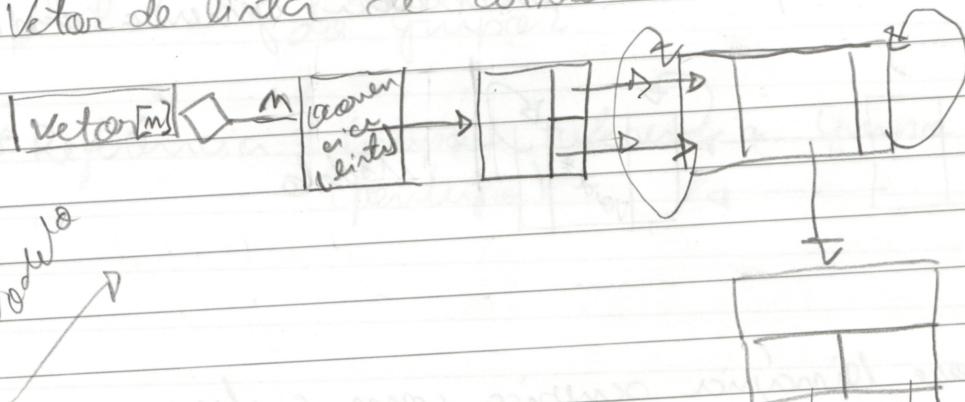
Se  $p_{carr} \rightarrow p_{prox} \neq NULL$

$p_{carr} \rightarrow p_{prox} \rightarrow p_{antes} = p_{antes}$

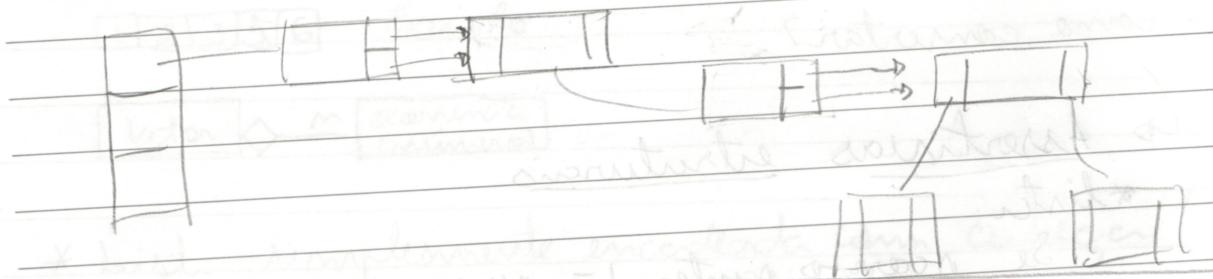
\* Árvore:

Ponteiro de um nó só subcarregar a esquerda nunca aponta para nem para o pai nem para o no de subcarregar a direita.

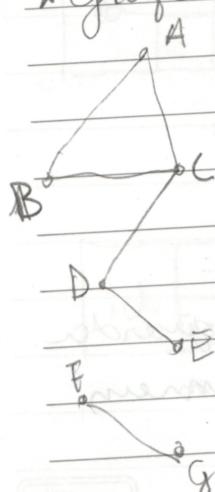
\* Vectors de llista de canvis



Exemple

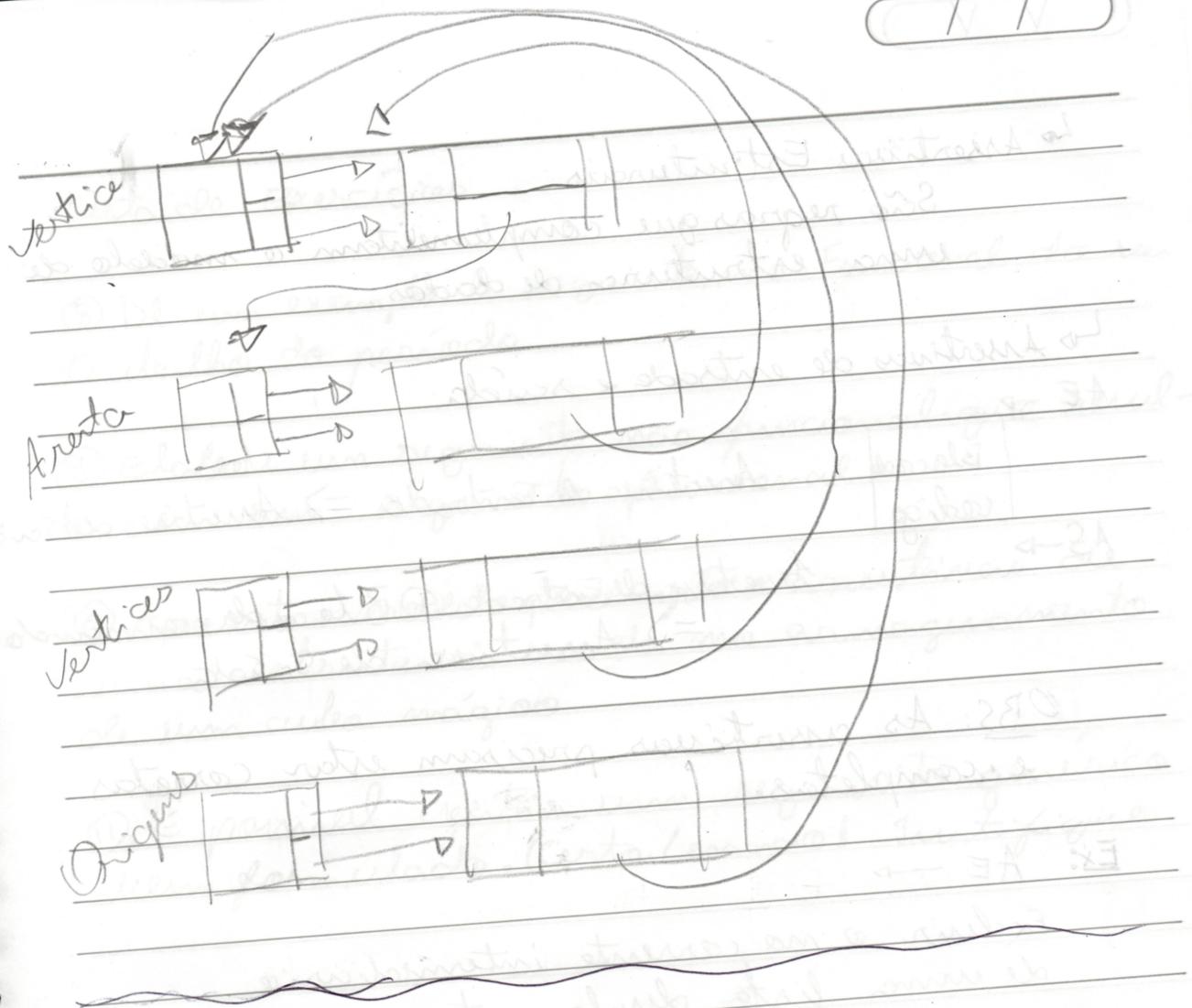


\* Grafo:



EX: (Democracia) llistes (origens, destinacions, alteracions, creacions, desaparicions)

Modelo:



### Assertivas

Regras consideradas válidas ao executar um determinado ponto do programa.



→ Assertiva → chegam aqui? Tudo OK  
ws print

São utilizadas em:

- argumentação de corretude
- instrumentação

→ código externo  
para verificar a corretude.

tilibra

1 / 1

### ↳ Assertivas Estruturais:

São regras que complementam o modelo de uma estrutura de dados

### ↳ Assertivas de entrada e saída:

AE →

Blco de  
código

Assertiva de Entrada  $\Rightarrow$  Assertiva de Saída

AS →

Assertiva de Entrada  $\oplus$  Blco de comando  
 $\Rightarrow$  Assertiva de Saída

OBS: As assertivas precisam estar corretas e completas.

Ex: AE →

Excluir o no corrente intermediário de uma lista duplamente encadeada com cabeça

AS →

AE = - lista existe e possui pelo menos 3 nós

- Ponteira corrente aponta para o nó intermediário que deve ser excluído
- Valem as assertivas estruturais da lista duplamente encadeada com cabeça

AS = - O nó foi excluído

- Valem as assertivas estruturais da lista duplamente encadeada com cabeça
- Ponteiro corrente aponta para o próximo nó da lista

## Implementação da Programação Modular

### ① Espaço de Dados

- São áreas de armazenamento

- Alocadas em um bloco

- Possui um tamanho

- Possui 1 ou mais nomes de referência

Ex.:  $A[7] \rightarrow$  7-º elemento do vetor A

\*ptAux  $\rightarrow$  Espaço apontado pelo ponteiro ptAux

\* ptAux  $\rightarrow$  Espaço que contém um endereço de memória

\* ptElemTaleSimb \* alerElemTaleSimb  
(char \* ptSimbolo)

{(\* alerElemTaleSimb char \* ptSimbolo)). Id

ou

alerElemTaleSimb (char \* ptSimbolo)  $\rightarrow$  Id

► O Subcampo id da estrutura apontada pelo ponteiro retornado pela função  
alerElemTaleSimb

## ② Tipos de dados

- Determinam:

- organização

- codificação

- Tamanho em bytes

- Compre de valores permitidos

### ① Organização:

Como interpretar o espaço de memória

### ② Codificação:

Definição da representação de dados

$\begin{array}{c} XX \\ \diagup \quad \diagdown \\ \text{dia} \end{array}$     $\begin{array}{c} XX \\ \diagup \quad \diagdown \\ \text{mes} \end{array}$     $\begin{array}{c} XX \\ \diagup \quad \diagdown \\ \text{ano} \end{array}$

### ③ Tamanho de bytes

Tamanho do tipo

### ④ Compre de valores permitidos

++

OBS: Um espaço de dados precisa estar associado a um tipo para que possa ser interpretado pelo programa desenvolvido em uma linguagem de programação.

OBS2: Tipos de tipos

Tipos computador

11

## 0352: Tipos de tipos

### - Tipos computacionais

(int, char, float, int\*, ...)

### - Tipos léxicos

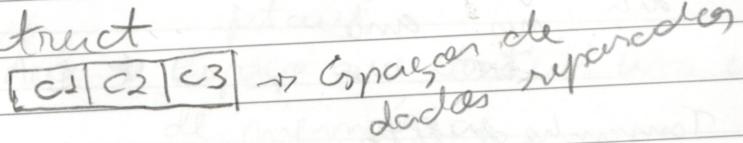
struct, union, enum, typedef

### - Tipos abstractos de dados

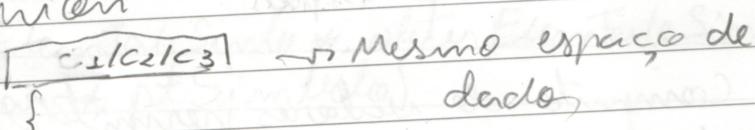
Estrutura encapsulada acessada  
pelo cliente unicamente por funções  
de acesso

## ③ Tipos Básicos

### \* Struct



### \* Union



int c1;

float c2;

char c3;

}

### \* Enum → Tipo enumerado

{

a, 1

b, 2

c, 3

d

tilibra

}

111

\* Typedef

↳ Typedef float tp\_veloc;

↳ Typedef float tp\_tempo;

Tp\_tempo tempo;

tp\_veloc veloc;

## ④ Declaração e definição de elementos

- Definir:

Aloca espaço de dados e armazena espaço  
ao nome (Binding)

- Declarar:

Atividades tipo de espaço.

Ex: int a → define e declara

struct → Declarar

mais... → Define

## ⑤ Declarações

### ① Implementação em C e C++

a) Declarações e definições de nomes  
globais exportadas pelo módulo  
remoto

Ex: int a;

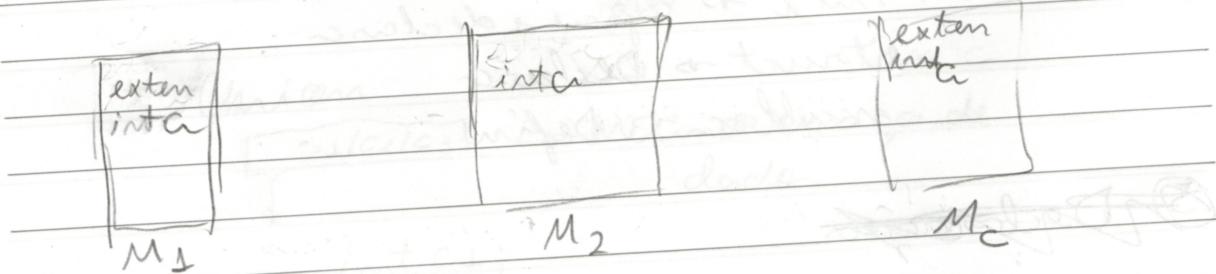
1 /

ii) ~~Declaracões~~ Declarações externas contidas no módulo cliente que somente declaram o nome sem associação a um espaço de dados

Ex: extern int a;

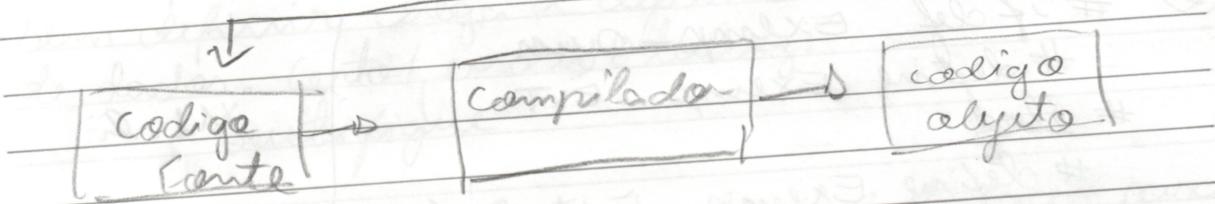
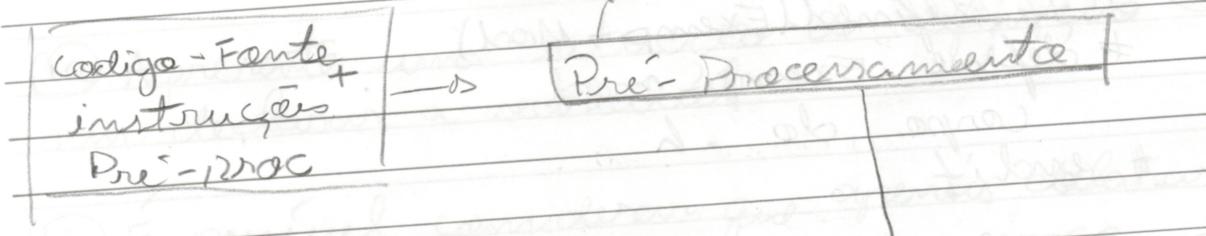
c) Declarações e definições de nomes globais encapsulados no módulo

Ex: static int a;



## ⑥ Pré-Processamento

super tipo  
#include



Exemplos de instruções para Pré-processador

- # define nome valor
- # include <nome.arquivo>  
"nome.arquivo" → Path
- # if defined ( nome ) ou # ifdef nome  
text V
- # if !defined  
ou
- # else  
text F
- # endif
- # undef nome

1 / 1

Ex bloco de código:

① # if !defined(Exemp-Mod)  
# define Exemp-mod  
corpo do .h  
# endif

② #if def Exemp-own - -  
#define Exemp-EXT } M1.h  
#else  
#define Exemp-EXT extern  
#endif  
Exemp-EXT int vector[7];  
#if defined(Exemp-own)  
= {1,2,3,4,5,6,7};  
#Else  
:  
#endif

Preprocessador

#define Exemp-own  
#include "M1.h"  
#undef Exemp-own

int vector[7]  
= {1,2,3,4,5,6,7},

#include "M1.h"

extern int vector[7];

11

## Estrutura de Funções

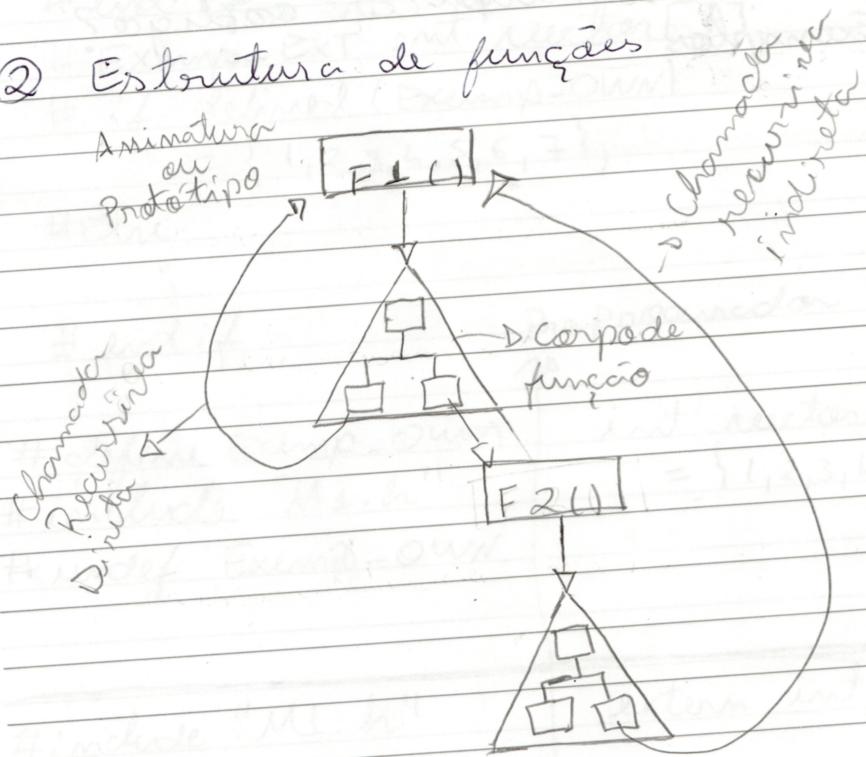
### ① Paradigmas

- Forma de programar
- ↳ Procedural

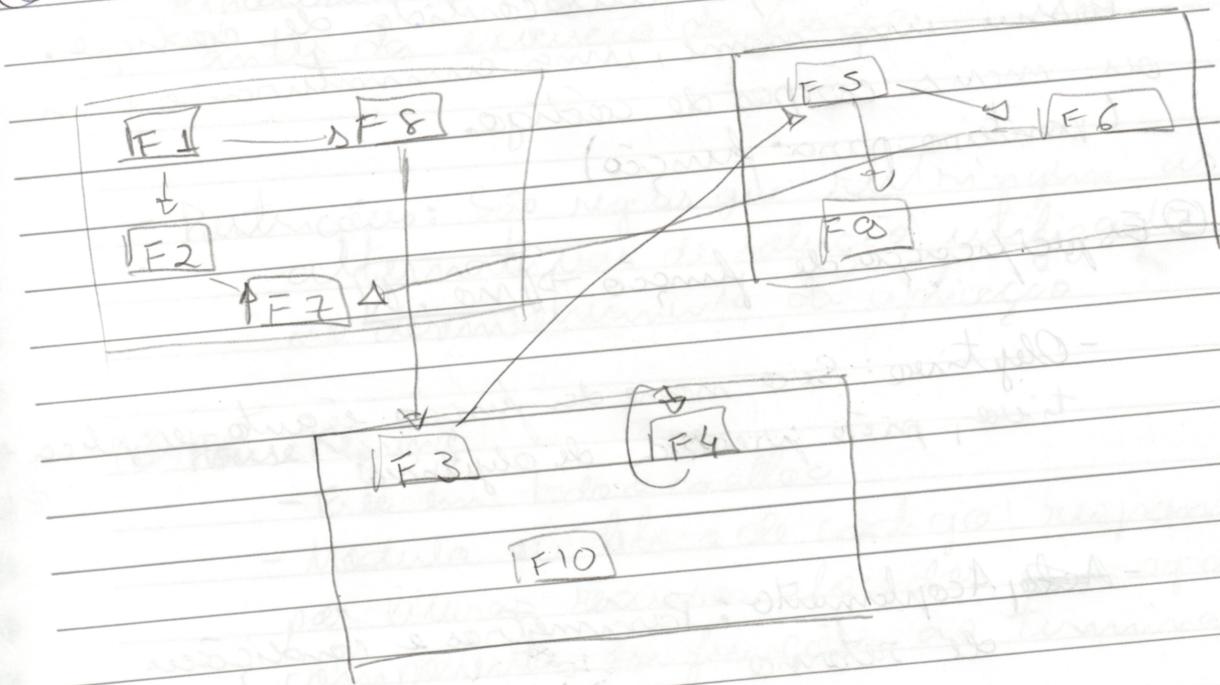
### ↳ Orientação a objeto

- P.O.O.
- Programação modular

### ② Estrutura de funções



### ③ Estrutura de chamadas



Arcos de chamada

$F_4 \rightarrow F_4 \rightarrow$  Chamada Recursiva Direta

$F_9 \rightarrow F_8 \rightarrow F_3 \rightarrow F_5 \rightarrow F_9 \rightarrow$  Chamada Recursiva indireta

$F_{10} \rightarrow$  Função morta neta estrutura modular

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

$F_1 \rightarrow$  Origem

1 /

#### ④ Função

É uma porção autocontida de código.  
Possui um nome, uma estrutura e um ou mais corpos de código.  
(ponteiro para função)

#### ⑤ Especificação de função → no.h

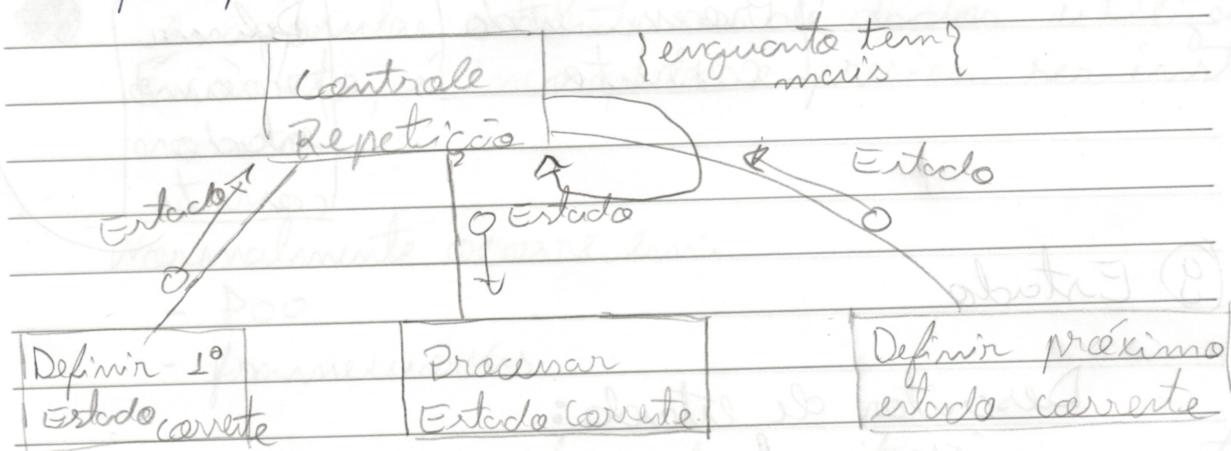
- Obrigatória: Se o nome da função é auto-explicativa, não precisa de especificação.
- Auxiliar/Acompanhamento: Parâmetros e condições de retorno.
- Condições de acoplamento: Averiguaçõeas de entrada e saída.
- Interface com o usuário: Qualquer mensagens, imagens, textos devem ser especificados.
- Requisitos específicos da funcionalidade da função.
- Ex: se o ponteiro é NULL, então ...

- **Hipóteses:** São regras que devem estar corretas antes da execução da função.
- **Restrições:** São regras que restringem as alternativas de solução utilizada no desenvolvimento da aplicação.

## ⑥ House Keeping

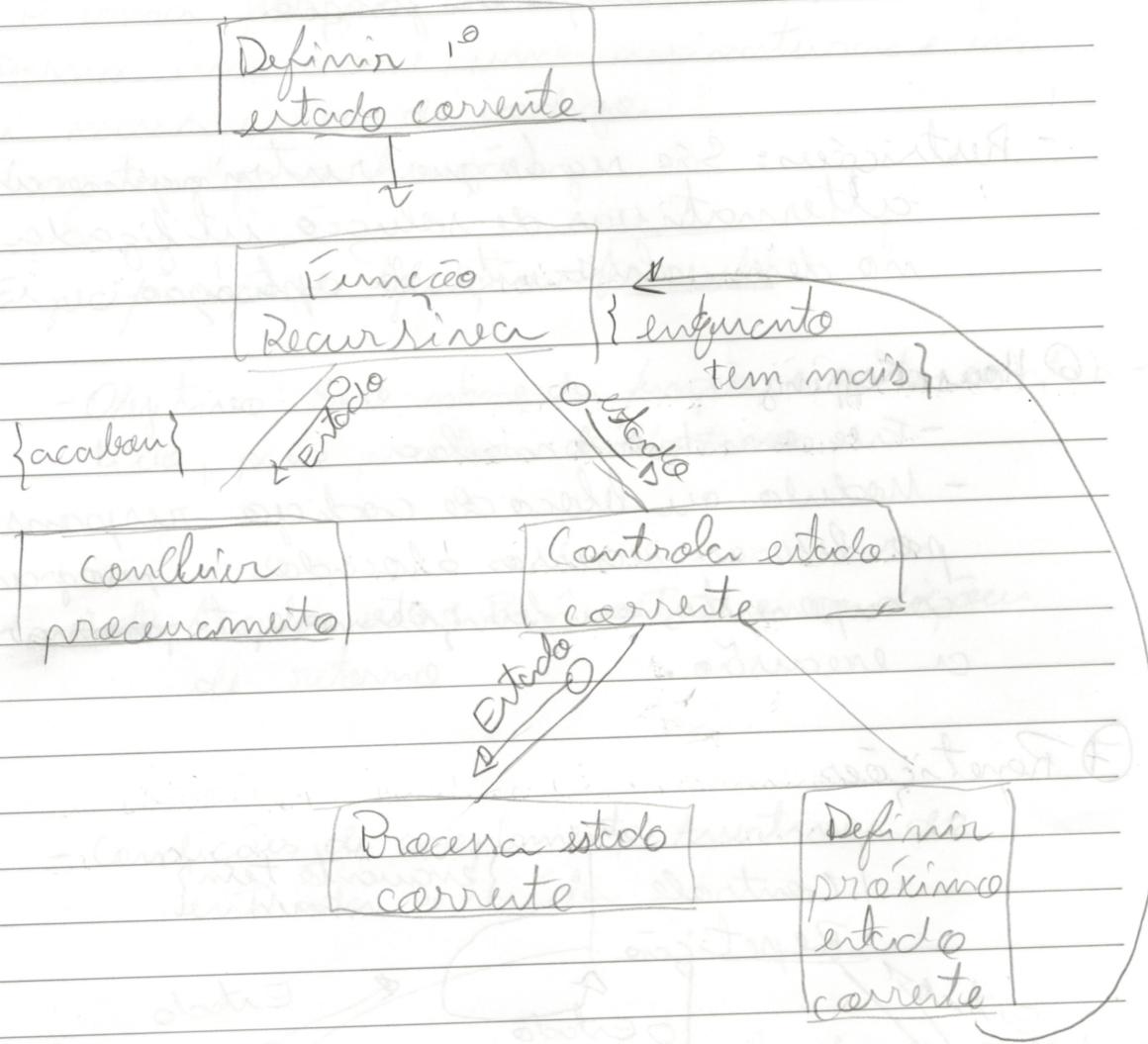
- Free em todo o malloc
- Módulo ou bloco de código responsável por liberar recursos alocados a programas, componentes ou funções ao terminar a execução.

## ⑦ Repetições



1 /

## ⑧ Recursão



## ⑨ Estado

Descriptor de estado:

- Variável ou variações que definem um estado

ex: busca sequencial  $\rightarrow$  int i;

busca binária  $\rightarrow$  int inf; int sup;

Estado:

- Valoração do descritor de estado

## 10 Esquema de algoritmo

Framework

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

~~OBS:~~ Esquema de algoritmo permitem  
encapsular estruturas de dados utilizada.  
É correto, é incompleto e precisa ser insta-  
niado

Normalmente ocorre em:

- POO
- frameworks

Se esquema correto e não pega com orienta-  
ções válidas nenhô proponho correto.

1 / 1

## ⑪ Parámetros de tipo puntero para función

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

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

```
int processArea (float valor1, float valor2,
                 float (*func)(float, float))
{
    printf ("%f", func(valor1, valor2));
}
```

```
condRet = processArea (5, 2, areaQuad);
```

```
condRet = processArea (3, 2, areaTri);
```