1 - Introdução

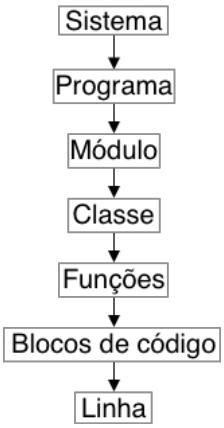
- Vantagens da programação modular:
  - Vencer barreiras de complexidade
  - Distribuição de tarefas em grupo
  - Reuso
  - Criação de um acervo de módulos reutilizáveis
  - Permite trabalhar com baselines de módulos testados
  - Desenvolvimento incremental (testes de módulos pós produção individual)
  - Aprimoramento individual (melhoramentos em módulos individualmente)
  - Reduz o tempo de complicação

2 - Princípios de Modularidade

- Módulo:
  - Física: unidade de complicação independente
  - Lógica: possui um único objetivo

- Elementos de programa:
  - Podem ser: blocos de códigos, fragmentos de texto de documentação, funções, figuras ou diagramas, seções de documentação, tipos de dados, classes, componentes...
  - Artefato: algo elaborado durante um processo de desenvolvimento e que possua identidade própria (algo que consegue ser versionado no desenvolvimento de uma aplicação)
  - Construto: uma versão de uma aplicação que consegue ser executada, mesmo que incompleta (build)

Hierarquia (em ordem de prioridade) - níveis de abstração:



3 - Interface

- Definição: Mecanismo de troca de dados, comandos e eventos entre elementos de programa. Responsável pela comunicação entre módulos.

(Obs: a interface sempre ocorre entre elementos de programa de mesmo nível de hierarquia)

- Formas de interface:
  - Arquivo: entre sistemas
  - Funções de acesso e seus parâmetros: entre módulos
  - Variáveis globais: entre blocos de código
  - Fornecida por terceiros: 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.
  - Em detalhe:
    - Sintaxe: regras
    - Semântica: significado

Exemplo de Interface:

```
tpDadosAluno * obterAluno(int id)
```

- Interface esperada pelo cliente:
  - Ponteiro válido referenciando os dados de aluno ou null

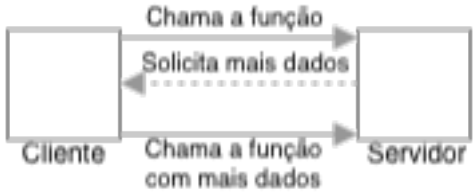
- Interface esperada pelo servidor:
  - Id válido
  - Acesso aos dados de alunos se o mesmo existir

- Interface esperada por ambos:
  - tpDadosAluno (interface fornecida por terceiro)

obs:
- Protocolo de uso: forma de se utilizar os itens que compõe uma interface para que esta possa ser operada corretamente

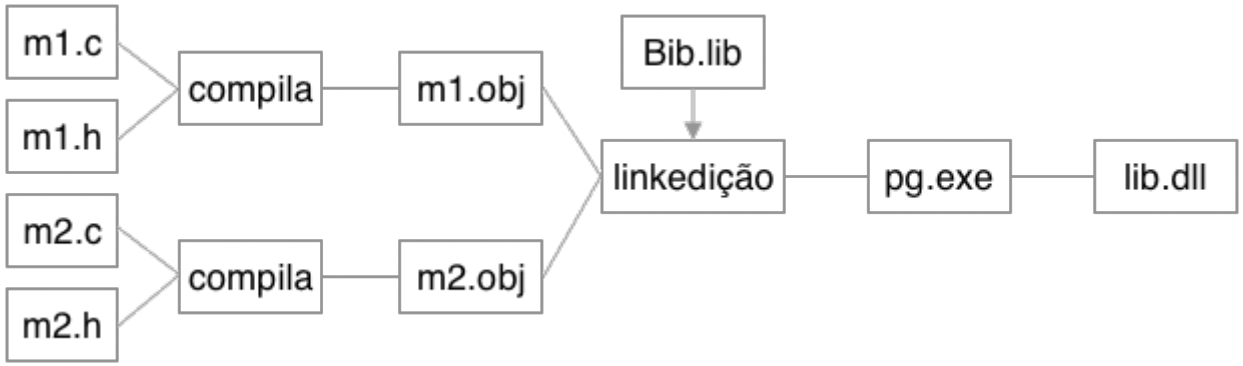
- Relacionamento cliente-servidor:
  - Módulos que conversam entre si para troca de dados. (função de acesso)
  - Caso especial callback: módulo servidor “pede” por mais dados para o módulo cliente, se tornando o módulo cliente temporariamente dentro do relacionamento anterior.

Caso Especial:
Suponha que um módulo M1 use uma função do M2. Sendo assim o M1 é cliente do M2. O M2 por sua vez retorna uma resposta assumindo um papel de servidor.



Call-Back: Quando M1 usa uma função do M2 e não fornece dados suficientes o M2 faz uma requisição para M1 pedindo mais dados. Dessa forma temporariamente o M2 está assumindo papel de cliente e o M1 servidor.

4 - Processo de Desenvolvimento



- Arquivos .c são os módulos de implementação
- Arquivos .h são os módulos de definição.
- a biblioteca.lib são bibliotecas estáticas.
- a lib.dll são bibliotecas dinâmicas.

5 - Módulo de Definição (.h)

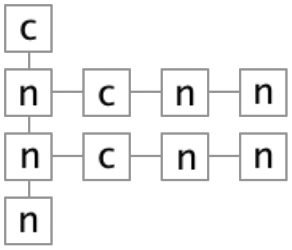
- Especificação externa voltada para os programadores do módulo cliente
- Protótipos ou assinaturas das funções de acesso
- Declarações e códigos públicos ao módulo (biblioteca.h)

6 - Módulo de Implementação (.c)

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

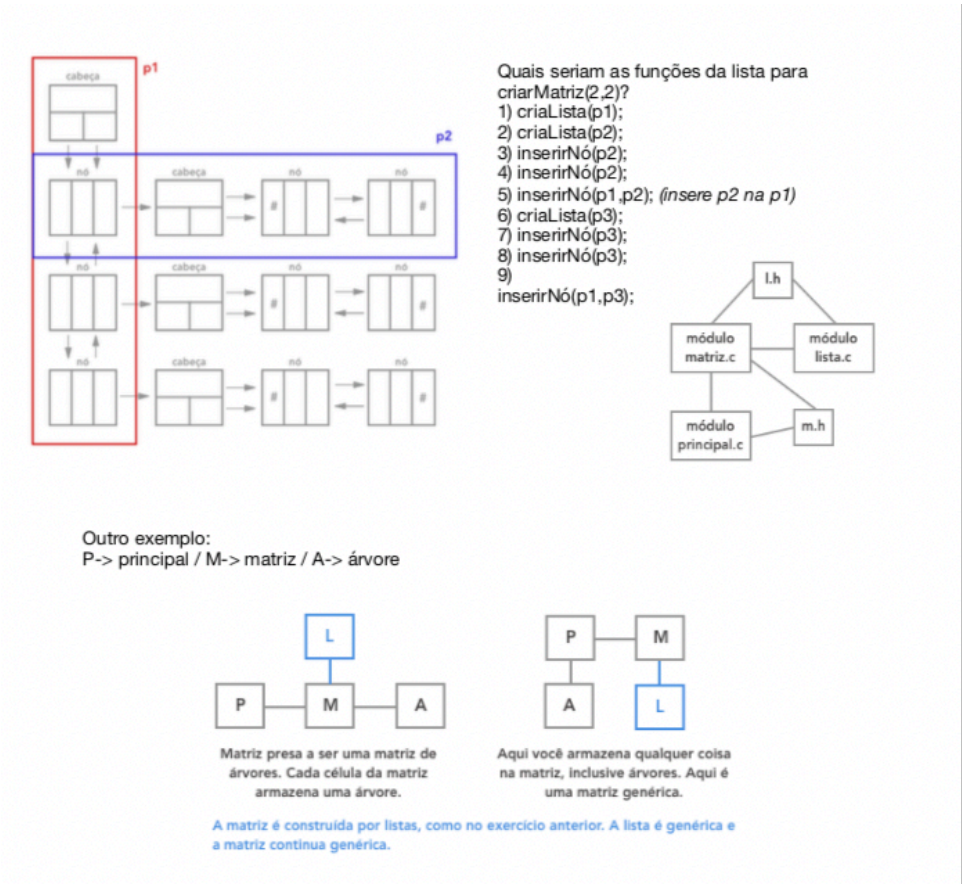
7 - Tipo Abstrato de Dados (TAD)

- Definição: a estrutura encapsulada que somente é conhecida pelos clientes através das funções de acesso
- ex.:



Onde:
c: são cabeças de listas
n: são os nós de lista

* Essa matriz é de 3x2 onde temos 3 nós de conteúdo
os 3 primeiros nós estão apenas ancorados na lista
Para modularizar algo desse tipo teríamos um modulo de "lista" para manipular os nós. Um modulo de "matriz" que por sua vez utiliza o modulo de lista. Por último o programa propriamente dito que utiliza a matriz. Note que não há necessidade do programa acessar a lista. Logo temos:



8 - Propriedades de Modularização

- Encapsulamento (proteção)
- Acoplamento (interface)
- Coesão (conceito)

9 - Encapsulamento

Vantagens do Encapsulamento:
- Manutenção facilitadas pois tudo relacionado a estrutura que está sendo encapsulada está dentro do mesmo módulo
- Facilita a documentação porque a estrutura encapsulada que se encontra documentada trata de um único conceito

Desvantagem: exagero do encapsulamento de conceitos.

Tipos de Encapsulamento:
- de código (ex: função de acesso de um módulo de implementação; não vista pelo módulo cliente, *for/while/if/switch,* função)
- de variáveis (static - classe, local - bloco de código, private - objeto, protected - estrutura de herança)
- de documentação (interna (.c), externa (.h) e de uso (para o usuário ex: leia-me))

10 - Acoplamento

Propriedade relacionada com a interface entre os módulos

- Conector: é o item da interface
  - ex.: protótipo de função, arquivo, variável global.

- Critérios de qualidade de acoplamento:
  - tamanho do conector (ex: função com 10 parâmetros vs função com 1 parâmetro)
  - quantidade de conectores (itens de interface realmente necesssarios e suficientes)
  - complexidade do conector

11 - Coesão

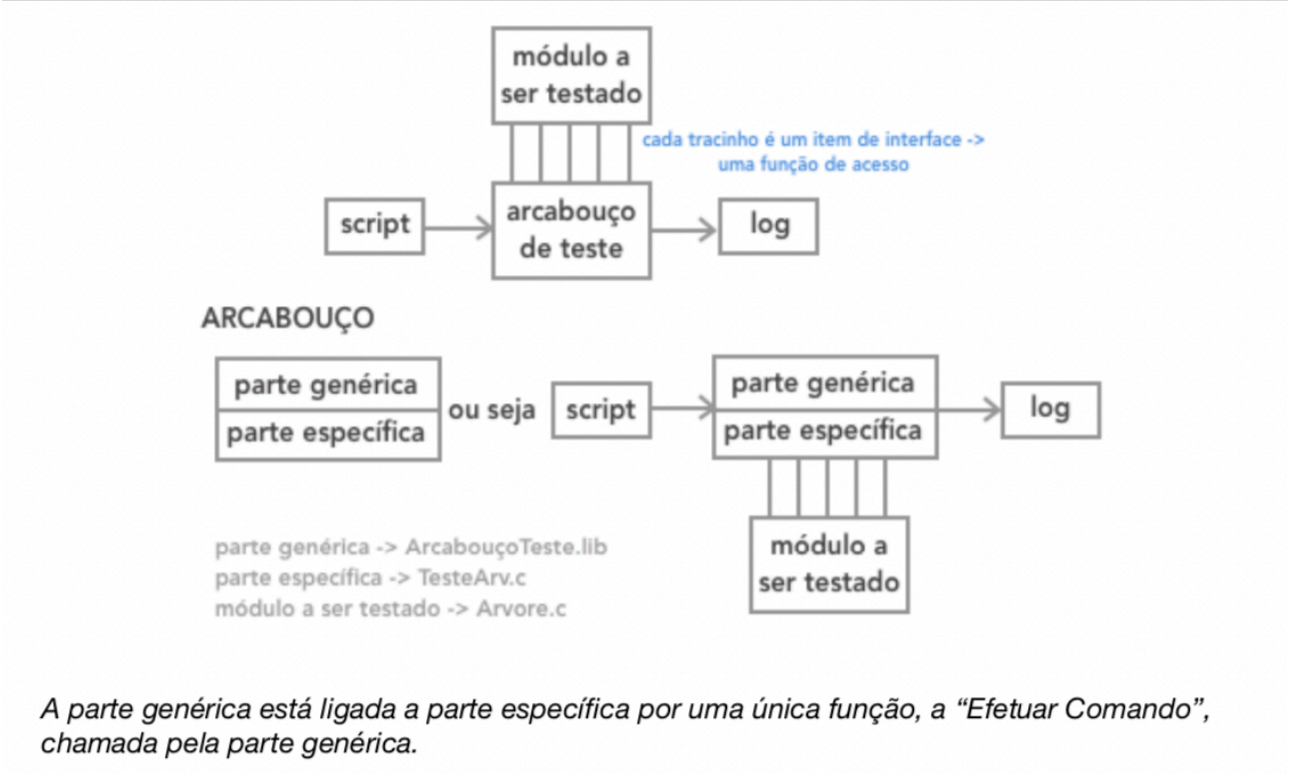
Propriedade relacionada com o grau de interdependência dos elementos que compõem o módulo (conceito)

- Níveis de coesão:
  - incidental: não há relação entre os vários conceitos inseridos no módulo
  - lógica: os elementos possuem uma relação lógica entre os conceitos de uma forma possivelmente genérica
  - temporal: os elementos estão relacionados pela necessidade de serem utilizados dentro do mesmo período de tempo
  - procedural: os elementos estão relacionados pela necessidade de serem executados numa mesma ordem (.bat)
  - funcional: elementos relacionados por funcionalidade
  - abstração de dados: um único conceito

12 - Teste Automatizado

Objetivo: Testar de forma automática um módulo recebendo um conjunto de casos de teste na forma de um script ordenado que gera uma log de saída com analise referente ao resultado esperado e o obtido.
* Testes automatizados são extremamente importantes para garantir que nada seja quebrado e passe desapercebido
* A partir do primeiro retorno esperado diferente do obtido no log de saída, todos os resultados de execução de casos de teste não são mais confiáveis.

- Framework de teste



A parte genérica está ligada a parte específica por uma única função, a "Efetuar Comando", chamada pela parte genérica.

- Script de Teste:
  - // comentário
  - == cado de teste -> testa determinada situação

- = comando de teste -> associado a uma função de acesso

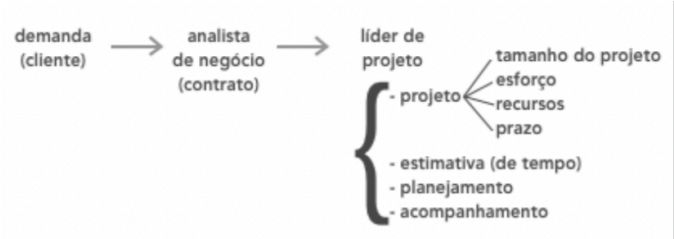
Obs: teste completo -> casos de teste para todas as condições de retorno de cada função de acesso do modulo (exceto de condição de retorno de estouro de memória).

- Log de Saída:
  - == caso 1
  - == caso 2
  - == caso 3
  - 1 >> função esperava 0 e retornou 1
  - 2 >>
  - 3 >>

    ...

Obs: == recuperar -> desfaz o último erro não previsto.

13 - Especificação de Requisitos



- Requisitos:
  - o que deve ser feito
  - nunca *como* deve ser feito

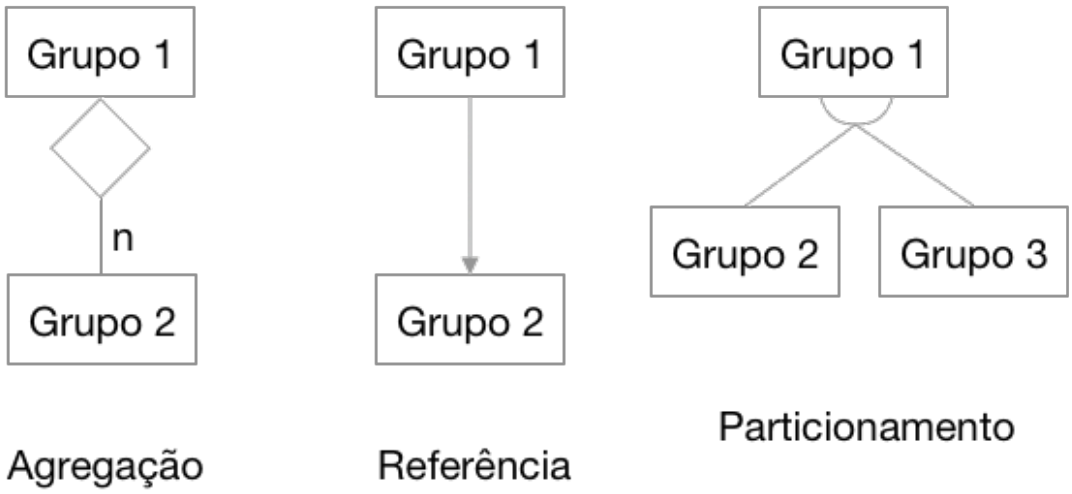
- Características:
  - curtos e diretos (sem “isto é”)
  - linguagem natural (a falta pode gerar requisitos ambíguos)

- Etapas da especificação:
  - elicitação (busca de informações)
  - técnicas:
    - entrevista
    - brainstorming
    - questionário
  - documentação
    - requisitos genéricos (escopo) e específicos (regras de negócio em níveis de funcionalidade ou programação)
  - verificação
    - análise da documentação para verificar compatibilidade
    - em conjunto com a equipe técnica
  - validação
    - cliente

- Tipos de Requisito:
  - funcional: regras que devem ser implementadas na aplicação relacionada com o negócio (objetivo da aplicação) - regras de negócio
  - não funcional: propriedades que a aplicação deve possuir e que não necessariamente estão relacionadas com o negócio (ex: login e senha - requisito de segurança; disponibilidade - 24x7; backup; velocidade - busca com resultados em no máximo 3 segundos)
  - inverso: o que a equipe não se compromete em fazer
  - exemplos:
    - bem formulado: para cada aluno deve ser cadastrado matrícula e nome
    - mau formulado: a interface deve ser de fácil utilização

14 - Modelagem de Dados

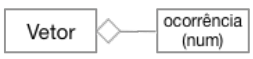
- Notação: UML



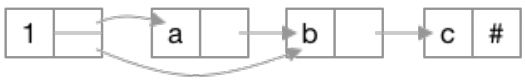
- Agregação - estrutura estática - ex. vetor
- Referência - estrutura dinâmica - ex. lista encadeada
- Particionamento ou Especialização

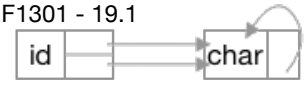
Ex.:

Vetor:



Lista simplesmente encadeada com cabeça:





Lista duplamente encadeada genérica com cabeça:

Árvore Binária genérica com cabeça:

- Assertivas Estruturais
  - Regra de Lista (duas regras de lista que não fazem parte da regra de árvore).
    se pCorr->pAnt != NULL pCorr->pAnt->pProx == pCorr
    se pCorr->pProx != NULL pCorr->pProx->pAnt == pCorr
  - Árvore
    Ponteiro de um nó de subárvore a esquerda nunca aponta para o par nem para nó de subárvore a direita.pAnt e pProx de um nó nunca aponta para o pai.

15 - Assertivas

- Definição: regras consideradas válidas ao executar um determinado ponto do programa.

- Utilização:
  - Argumentação de corretude: provar que determinado bloco de código está correto. Linha de raciocínio para mostrar que aquele bloco de código está funcionando, ou seja, gerar uma saída esperada para qualquer entrada que ele receba.
- Instrumentação: transformar assertivas em blocos de códigos para que eles se auto verifiquem (bloco de controle).Para gerar blocos de controle para o próprio código verificar se ele está correto ou não. Ex: printf

obs.: Devemos aplicar assertivas em trechos complexos onde a chance de erro é bem grande.
- Assertivas estruturais: são costuras que complementam um modelo de uma estrutura de dados.
- Assertivas de Entrada e Saída
  - Assertiva de entrada: deve estar verdadeira antes de entrar no bloco. Como os dados devem estar preparadas para o bloco utilizar. Ex: inicializar o índice antes do loop.

- Assertiva de saída: deve estar verdadeira na saída de execução do bloco. Como esses dados vão estar depois da execução daquele bloco.

Obs: As assertivas precisam estar corretas e completas. Cada função vai ter uma assertiva de entrada e uma assertiva de saída para ela.

16 - Implementação de Programação Modular

- Espaço de Dados:
  - são áreas de armazenamento alocadas em um meio
  - possuem um tamanho
  - possuem um ou mais nomes de referência
  - ex.:
    - A[j] -> j-ésimo elemento sequencial do vetor A
    - *ptAux -> espaço apontado por ptAux
    - ptAux -> espaço que contém um endereço de memória

- Tipos de Dados:
  - determinam organização, codificação, tamanho em bytes e conjunto de valores permitidos.
  - obs.: um espaço de dados precisa estar associado a um tipo para que possa ser interpretado pelo programa desenvolvido em linguagem tipada.
  - obs2.:
    - tipos de tipos:
      - tipo computacional: int, char, char *, ...
      - tipo basico: struct, enum, union, typedef
      - tipo abstrato de dados

- Tipos Básicos:
  - struct: [c1][c2][c3] -> int, char, int
  - union: { int c1, char c2, char c3}
  - enum: { a, b, c, d, e}

- Declaração e Definição de Elementos
  - Definir: aloca espaço de dados e amarra espaço ao nome (binding)
  - Declarar: atribui tipo ao espaço

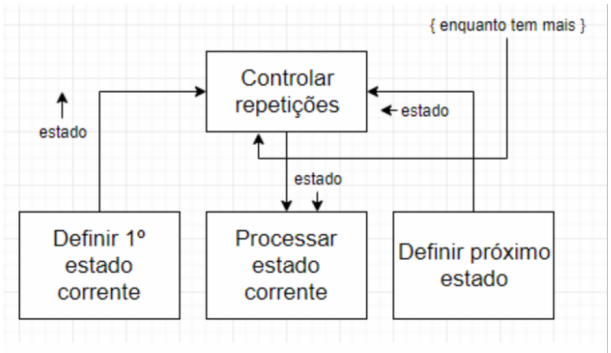
- Implementar em C e C++
  - declarações e definições de nomes globais exportados pelo módulo servidor. ex.: int a;
  - declarações externas contidas no módulo cliente e que somente declaram o nome sem associá-lo a um espaço de dados. ex.: extern int a.

17 - Estrutura de Funções

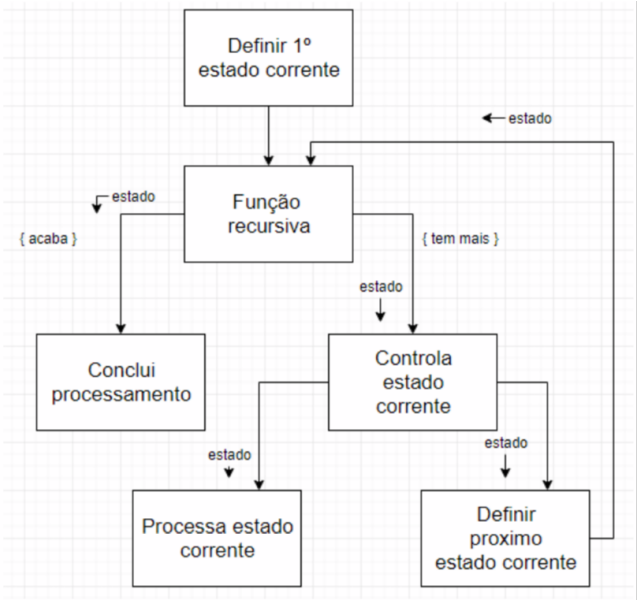
- Paradigma
  - Formas de programar
    - Procedural - passo a passo -> receita de bolo - criação de funções para não duplicar código. Programação estrutural (Pascal, C)
    - Orientada a Objeto - várias entidades envolvidas e a forma como elas se conectam. Conjuntos de informações que tenho que agrupar, os métodos que tenho que executar. - atributos (TAD) e os métodos
      - Programação orientada a objetos
      - Programação modular (mesmo utilizando uma linguagem não orientada a objetos. A gente pensa em entidades).

- Função - é uma porção auto contida de códigos. Pode precisar de algo da outra função, mas todo código que ela precisa tem.
  - Possui: um nome, uma assinatura e um ou mais corpos de código

- Especificação da função: fica no .h antes de cada função de acesso.
  - Objetivo: Dizer para o que serve a função. Se o nome da função é autoexplicativo esse item pode ser retirado.
  - Acoplamento
    - Parâmetros e condições de retorno (o que recebe e o que devolve). Tem a ver com a interface.
  - Condições de Acoplamento: Assertivas de entrada e saída
  - Interface com o usuário: Mensagens, imagens, textos, saídas em tela para o usuário
  - Requisitos: O que precisa ser feito.
  - Hipóteses: Regras consideradas básicas antes da definição da função. São regras pre definidas que assumem como válida alguma determinada ação ocorrendo fora do escopo, evitando assim o desenvolvimento de códigos desnecessários.
  - Restrições: Regras que restringem as alternativas de solução utilizadas em um desenvolvimento de uma aplicação. São regras que limitam a escolha das alternativas de desenvolvimento para uma determinada solução. Existem várias soluções para determinar determinado problema, mas as vezes algumas não pode usar por limitações da maquina, ou por falta de tempo, etc. São restrições de desenvolvimento. Ex: a função de mostrar vai usar só caracteres ASCII. Exemplo de restrição do trabalho: só pode ser feito em linguagem C.
  - House Keeping (Manutenção de casa - faxina): free em cada malloc. Módulo ou bloco de código responsável por liberar recursos alocados a programas, componentes ou funções ao terminar a execução.
  - Repetição:



- Recursão:



- Estado
  - Descritor de estado: variáveis (conjunto de dados) que definem um estado.
    Obs.: não precisa ser único (como é o caso na pesquisa binária)
    Obs. 2: não necessariamente é observável -> qual o estado da próxima linha de um arquivo de texto? Curso de posicionamento de arquivo.

- Esquema de Algoritmo

```
{
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;}
}}
```

Algoritmo de busca binária (limites inferiores e superiores)
Sublinhado = hotspot (onde se trata da parte específica)
O restante = parte genérica do algoritmo, o esquema de algoritmo.
O conjunto do hotspot com a parte genérica que se chama framework.
Esquemas de algoritmo permitem encapsular a estrutura de dados utilizada. É correto, independe de estrutura, é incompleto e precisa ser instanciado.
Normalmente ocorrem em: programação orientada a objetos, frameworks.
Se o esquema está correto e o hotspot com assertivas válidas então: programa correto.

- Parâmetros do tipo ponteiro para função

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

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

Ambas as funções possuem os mesmos parâmetros (retorna float e recebe dois floats).

```
int ProcessaArea(float valor1, float valor2, float (*Func)(float, float)) {
 ...

printf("%f", Func(valor1, valor2));
...
 }
```

Na função ProcessaArea um dos parâmetros é uma função
condRet = ProcessaArea(5, 2, areaQuad);
condRet = ProcessaArea(3, 3, areaTri);
Chamam a mesma função mas como o parâmetro mandam funções.