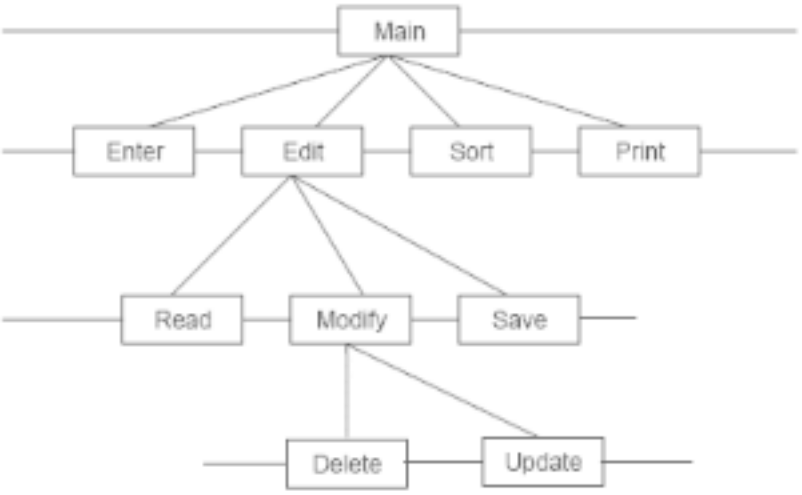


Anotações do Caderno de Programação Modular

Professor: Flávio Bevilacqua
Curso: **INF1031**
Período: 2018.2
Aluno: **Guilherme Dantas de Oliveira**

PUC RIO | DEPARTAMENTO DE INFORMÁTICA PUC-RIO

Índice

Aula 1

Bibliografia: “Programação Modular” de Arndt Von Staa
Os trabalhos estão organizados em:

- **T1: Arcabouço do Teste**
- **T2 e T3: Trabalho do Período**
- **T4: Instrumentação**

As notas dos trabalhos consistirá em 50% da nota do trabalho em si e 50% das anotações feita pelo aluno em sala de aula, em formato DIGITAL.

As provas P1 e P2 são com consulta, com reposição uma semana depois de cada aplicação. Em caso de a reposição também coincidir com alguma prova de outro curso, agendar com o professor outro horário e data para repor a prova. As datas das provas são 10/10 e 10/12.

Os graus são calculados pela seguinte média ponderada:

```
   G1 = ( T1 + 2*T2 + 2*P1)/5
G2 = (T3 + 2*T3 + 2*P2)/5
```

Introdução

★ **Vantagens de Programação Modular:**

- Vencer barreiras de complexidade
- Facilita o trabalho em grupo (paralelismo)
- Reúso de componentes
- Facilita a criação de um acervo ou coleção
- Desenvolvimento incremental: estratégia de desenvolvimento que secciona o projeto em partes a serem implementadas. Neste caso, os módulos.
- Aprimoramento individual de um módulo, sem precisar compilar o projeto todo.
- Facilita a administração de baselines: versões estáveis da build e as versões de cada módulo do projeto que funcionam seguramente.

Aula 2

★ **Módulo**

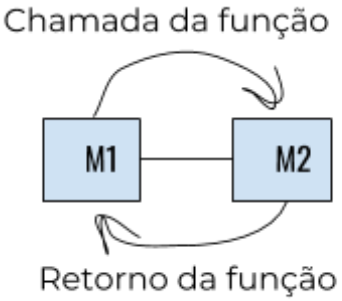
- Definição física: unidade de compilação independente (como um .c)
- Definição lógica: trata de um único conceito

★ **Abstração do Sistema**

- Abstrair é selecionar o que é necessário e o que não é necessário para o sistema. É o passo em que se define o **escopo**.
- Níveis de abstração: SISTEMA - PROGRAMAS - MÓDULOS - FUNÇÕES - BLOCO DE CÓDIGO - LINHAS DE CÓDIGO
- **Artefatos** são itens com identidade própria criados dentro um processo de desenvolvimento. O artefato pode ser versionado (ou seja, que participa de um controle de versão). Um arquivo é um exemplo de artefato.
- Construto ou **build**: resultado apresentável, mesmo que incompleto, do sistema.

★ **Interface**

- Definição: mecanismo de troca de dados, estados e eventos entre elementos de um mesmo nível de abstração.
- Entre sistemas: arquivos
- Entre módulos: métodos ou funções de acesso
- Entre funções: passagens de parâmetro
- Entre blocos de código: variável global
- **Relacionamento Cliente-Servidor:** o módulo M1 chama a função do módulo M2. Portanto, M1 é o cliente e M2 é o servidor desta relação.

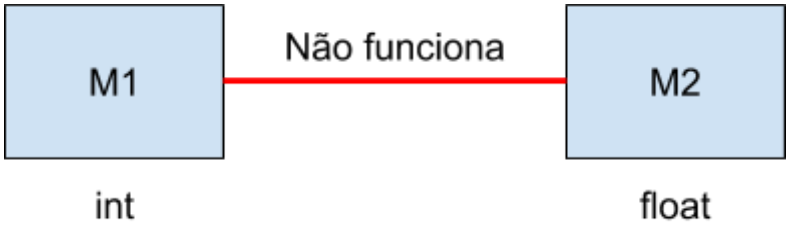


- **Caso especial**: o **callback** é quando o cliente faz uma chamada de função e o servidor requer mais dados do cliente para retornar a função. Nesse momento, há uma inversão de papéis entre os módulos.
- **Interface oferecida por terceiros**: ambos M1 e M2 usam o mesmo tipo de struct “aluno”. Para evitar futuras adversidades da duplicidade de dados, vincula-se a estes módulos um arquivo header para padronizar definições de structs, funções e variáveis destes módulos.

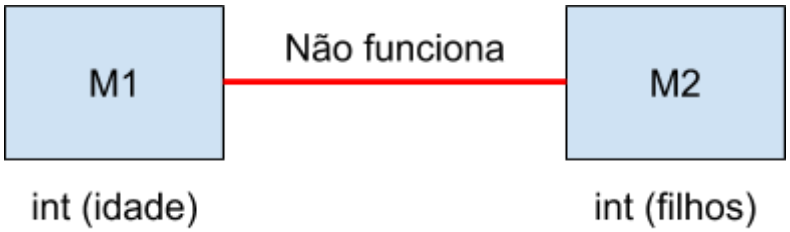
Aula 3

- **Interface em detalhe:**

■ Sintaxe: regras para uma troca de dados do mesmo tipo. (uma função não pode receber valores de tipos distintos, como int e float)



■ Semântica: do mesmo domínio de valores (significado). (O retorno de uma função deve estar bem definido no seu sentido. O que retorna?)

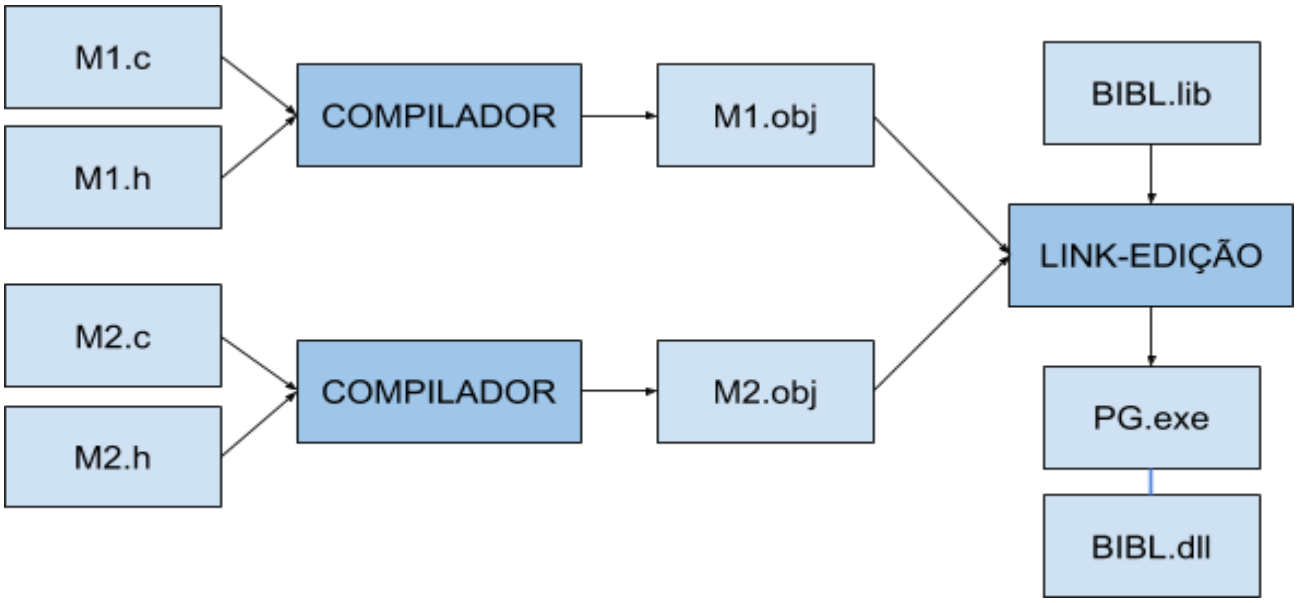


- **Análise de Interfaces:**

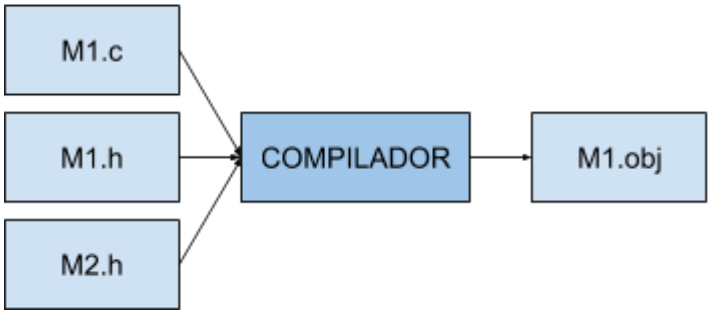
```
tpDadosAluno *obterDadosAluno(int mat);
```

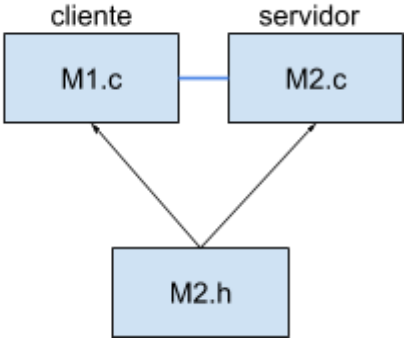
■ Interface esperada pelo cliente: um ponteiro para dados do aluno correto ou NULL.
■ Interface esperada pelo servidor: um inteiro válido representando a matrícula do aluno.
■ Interface esperada por ambos: a struct tpDadosAluno

★ **Processo de Desenvolvimento:** o processo por trás da geração de um constructo se dá através do seguinte fluxograma…



Na compilação de um dado módulo, todas as interfaces que tal módulo emprega serão “convocadas” na etapa em que se compilam os módulos, resultando num único objeto. Veja abaixo a ilustração deste processo:





Cada item deste fluxograma tem um nome particular:

.c	Módulo de Implementação
.h	Módulo de Definição
.lib	Biblioteca Estática
.dll	Biblioteca Dinâmica

★ **Bibliotecas estáticas e dinâmicas:** as vantagens e desvantagens de cada biblioteca está expressa na seguinte tabela…

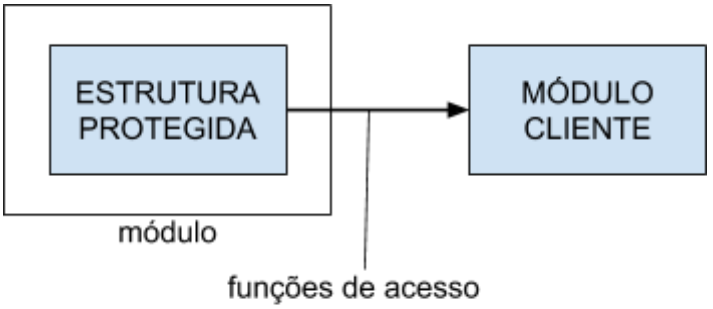
	Estática (.lib)	Dinâmica (.dll)
VANTAGEM	A .lib em tempos de link-edição já é acoplada à aplicação executada.	Só é carregada uma instância de biblioteca dinâmica mesmo que várias aplicações a acessem.
DESVANTAGEM	Existe uma cópia desta biblioteca estática para cada executável armazenada que a utiliza.	A .dll precisa estar na máquina para a aplicação funcionar.

★ **Módulo de definição (.h):**
- Interface do módulo
- Contém protótipos das funções de acesso
- Interface fornecida por terceiros
- Documentação voltada para o programador do módulo cliente

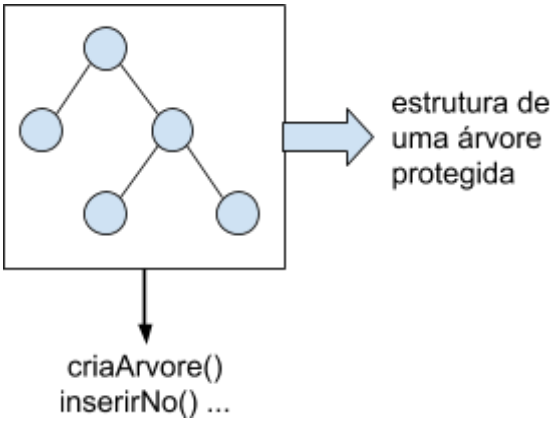
★ **Módulo de implementação (.c):**
- Código das funções de acesso
- Códigos e protótipos das funções internas
- Variáveis internas ao módulo
- Documentação voltada para o programador do módulo servidor

Aula 4

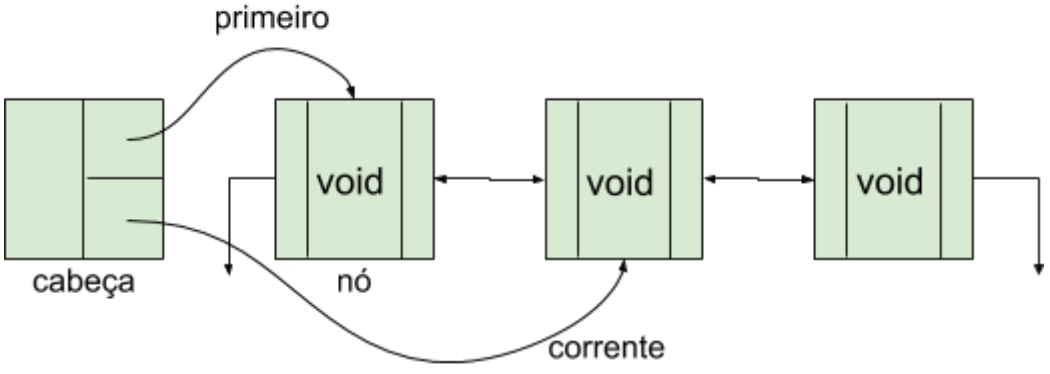
★ **Tipo abstrato de dados (TAD):** é uma estrutura encapsulada em um módulo que somente é conhecido pelos módulos clientes através das funções de acesso disponibilizada na interface.



No nosso caso, a estrutura protegida se trata de um TAD de Árvore, e as funções de acesso manipulam estes dados internos…



❑ **Exemplo:** Num TAD de Lista há duas structs: Nó e Cabeça. Tudo que fizermos dentro do módulo será através das funções de acesso.



Para administrar um TAD Lista, usamos funções de acesso como:

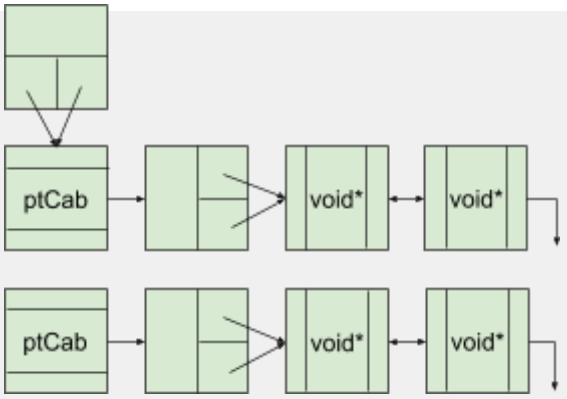
```
  ir_prox(ptCab plista);
ir_ant(ptCab plista);
obter(ptCab plista);
criarLista(ptCab *plista); (1)
inserirNo(ptCab plista, void *conteudo); (2)
```

Funções como a (1) empregam a chamada **passagem de parâmetro por referência**: o endereço de uma variável no módulo cliente que terá seu valor alterado dentro da função de acesso. No caso desta função, o parâmetro é o endereço da struct Cabeça (apontará para a cabeça que terá posse da lista criada).

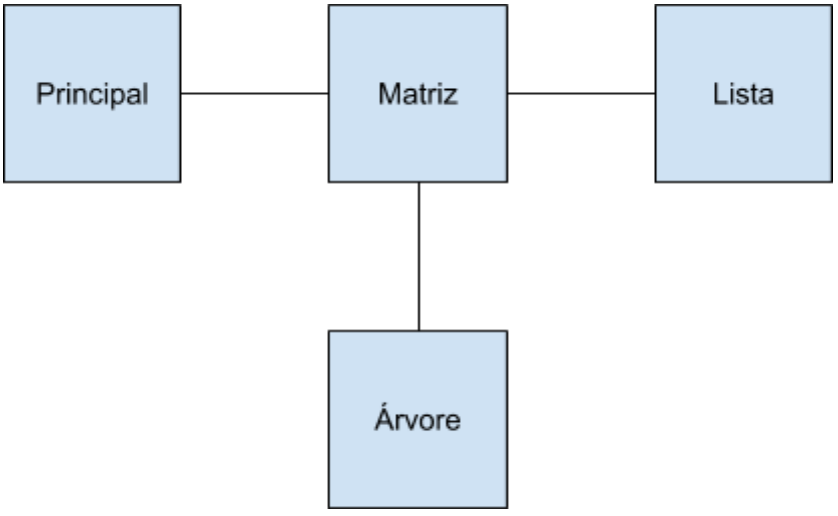
Já funções como a (2) empregam a chamada **passagem por valor**: acessamos o valor da variável para utilizá-la internamente, sem alterar o seu valor. No caso, o struct tpCab* é simplificado em ptCab (através de #typedef). Dentro do módulo servidor, há acesso permitido para mexer na estrutura protegida.

Assim, podemos montar uma matriz 2x2 por meio do TAD de Lista:

```
  criaLista(p1);
criaLista(p2);
inserirNo(p2,NULL);
inserirNo(p2,NULL);
inserirNo(p1,p2);
criaLista(p3);
inserirNo(p3,NULL);
inserirNo(p3,NULL);
inserirNo(p1,p3);
```

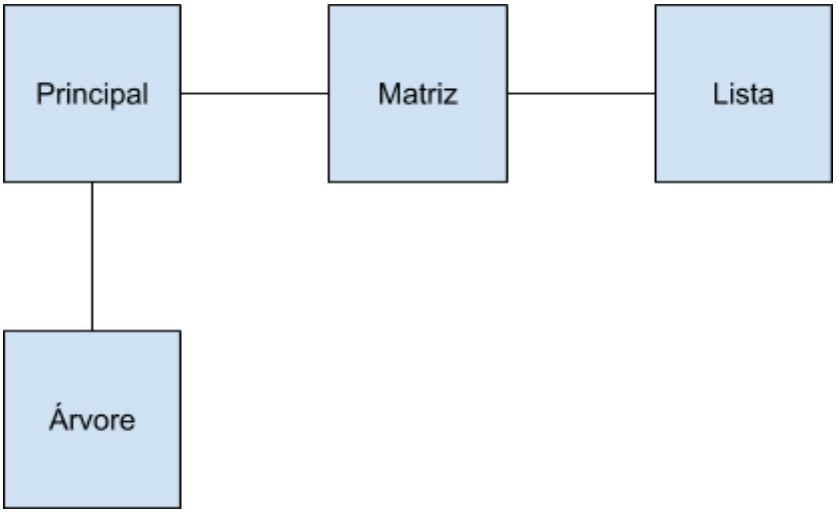


Deve-se ressaltar que, a partir do momento em que reciclamos o código de Lista na implementação do módulo Matriz, o módulo principal não precisa fazer chamadas de funções de acesso do módulo Lista, pois o módulo Matriz internamente faz tais chamadas.



No grafo ao lado, podemos concluir que o módulo Matriz genérico necessita de Lista para sua estrutura, e que pode armazenar árvores em cada elemento. Contudo, este modelo impossibilita que o módulo principal crie

árvores fora de uma matriz.



Já neste modelo, podemos perceber que é possível criar Árvores e Matrizes individualmente, mas também Matrizes de Árvores. A implementação de Matrizes cujos elementos são árvores cabe ao módulo Principal, contudo.

★ **Encapsulamento**: propriedade relacionada com a proteção dos elementos que compõem o módulo.

- Objetivo: facilitar a manutenção e impedir a utilização indevida da estrutura do módulo.

- Outros tipos de encapsulamento: de documentação

  - Doc. interna: .c (módulo de implementação)
  - Doc. externa: .h (módulo de definição)
  - Doc. de uso: manual de usuário

- De código - blocos de código visíveis apenas:

  - Dentro de módulo
  - Dentro de outro bloco de código (por exemplo: conjunto de comandos dentro de um loop)
  - Código de uma função

- De variável:

  - Private: encapsulamento no objeto
  - Public: global na Programação Orientada a Objetos
  - Global: global fora de Programação Orientada a Objetos
  - Global Static: no módulo
  - Protected: estrutura de herança (classes filhas herdam)
  - Static: global à classe (Programação Orientada a Objetos)

Aula 5

★ **Acoplamento:** propriedade relacionada com a interface entre os módulos.

- Conector - item de einterface. Ex: funções de acesso, variável global.

A qualidade de uma interface é medida pela:

- Quantidade de conectores - tudo na interface é necessário? É suficiente para que o código funcione?
- Tamanho do conector - quantidade de parâmetros de uma função, que pode ser minimizada ao agruparmos as variáveis em dados estruturados (struct de Endereço, de Data etc).
- Complexidade do conector - explicar na documentação, utilizar mnemônicos.

OBS: um bom acoplamento é o desejável; um alto acoplamento é quando as funções de acesso são altamente entranhadas com parâmetros.

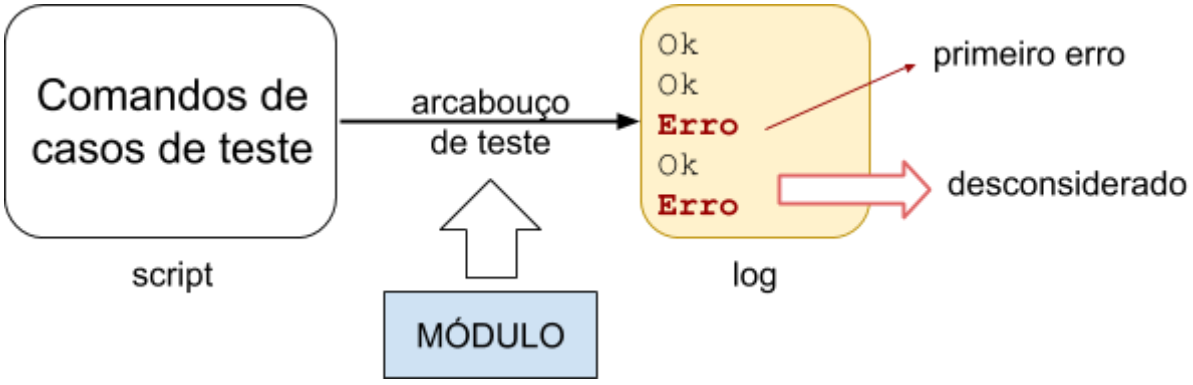
★ **Coesão:** propriedade relacionada com o grau de interligação dos elementos que compõem um módulo. Está diretamente ligada com a integridade do conceito de um módulo. Os níveis de coesão são…

- Incidental - pior coesão é quando dois conceitos que não tem qualquer semelhança estão no mesmo módulo.
- Lógica - elementos logicamente relacionados com uma ideia por trás (processa, calcula…).
- Temporal - itens que funcionam num mesmo período de tempo ou etapa (de_para, cancelar_matrícula…)
- Procedural: itens que são acionados em sequência / ordem lógica (linhas de um arquivo Batch ou .bat).
- Funcional: funções semelhantes (GeraRelatórioReceita, GeraRelatórioQualidade…)
- Abstração de dados: trata de um único conceito (TAD)

Aula 6

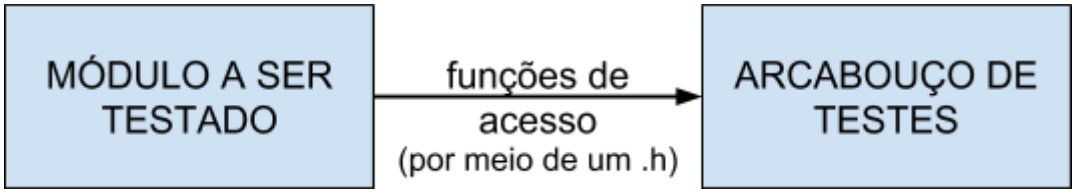
Teste Automatizado

★ **Objetivo:** é um script que faz o trabalho de testar de forma automática um módulo, recebendo um conjunto de casos teste e gerando um log de saída com a análise entre o resultado esperado e o objetivo.



- OBS: a partir do primeiro retorno esperado diferente do obtido no log de saída, todos os resultados de execução de casos de testes não são mais confiáveis. Isto é, podem haver falsos positivos e erros que dependem de erros.

★ **Framework de teste:** o arcabouço de teste possui uma parte genérica e uma parte específica



- Genérica: lê o script e gera o log
- Específica: traduz o script para funções de acesso do módulo a ser testado.
- No caso: a parte genérica está no Arcabouçoteste.lib e a específica está no TESTARV.C. O único conector entre estes dois é a função EfetuarComando, que é chamada pela parte genérica, onde a entrada é o comando e a saída é o retorno da função a ser testada.

★ **Script de Teste:** uma linguagem com os seguintes comandos…

- // é comentário.
- **==** é caso de teste: deseja testar um comando em um determinado contexto geral. Exemplo: excluir o nó raiz, excluir o nó intermediário, excluir nó final (folha).
- **=** é comando: chama uma função de acesso, sucedido de argumentos
- OBS: um teste completo é aquele que testa todas as condições de retorno de cada função de acesso do módulo (menos o estouro de memória).

★ **Log de Saída:** uma linguagem com as seguintes mensagens…

- ESPERADO COINCIDE COM OBTIDO: ==caso1
- ERROS NÃO ESPERADOS: 1 >> Função esperava 0 e retorno 1
- ERROS ESPERADOS: 1>> 0>>
  - Graças à função recuperar, zeramos o contador de erros caso quisermos testar o arcabouço de teste, não contabilizando como um erro não-previsto.

★ **Parte Específica:** A parte específica que necessita ser implementada para que o framework (arcabouço) possa acoplar na aplicação chama-se hotspot. Ex: TESTARV.C. A única função deste módulo é EfetuarComando, que recebe um comando de teste e executa com módulos a serem testados, retornando se o valor obtido coincide com o esperado ou se a estrutura do script está errada (para parte genérica).

Aula 7

Processo de Desenvolvimento em Engenharia de Software

★ A **demanda** vem do **cliente** para a empresa na figura do **analista de negócios**, que trata dos contratos, e define um **líder de projeto.**

○ O Líder de projeto cria o projeto, isto é, o **esforço**, os **recursos**, o tamanho (ponto de função) e o **prazo**. Para isso, ele dá uma estimativa (**deadline**). É ele quem vai planejar o projeto e supervisionar / acompanhar o seu desenvolvimento no decorrer do processo.

★ O processo é delineado pelas seguintes etapas:

A. Requisitos
a. Elicitação
b. Documentação
c. Verificação
d. Validação

B. Análise e projeto
a. Projeto lógico: modelagem de dados em M.E-R, UML...
b. Projeto físico: entidades como tabelas, módulos, arquivos...

C. Implementação
a. Programas
b. Teste Unitário: caso não apresente nenhum erro, passa para a próxima fase.

D. Testes
a. Tese Integrado - caso haja erros nesta etapa, o projeto retrocede ao item C.

E. Homologação: o cliente faz uso da aplicação e julga erros ou modificações (feedback), que será considerada no retrabalho (custeado).
a. Erros
b. Sugestões

F. Implantação
a. Gerência de configuração
b. Qualidade de software

Aula 8

Especificação de Requisitos

★ **Definição de Requisito:** O que tem que ser feito, não COMO deve ser feito. Nesta fase não compete a nós pensar em especificações técnicas, como estruturas de dados, funções de acesso, módulos etc.

★ **Escopo de Requisito:** O que precisa e o que não precisa, a abstração do projeto. Há requisitos mais genéricos e mais específicos.

★ **Fases da Especificação:**

- **Elicitação:** captar informações do cliente para realizar a documentação do sistema a ser desenvolvido. As técnicas de elicitação mais comumente usadas são entrevistas, brainstorm e questionário, sendo este último muito utópico, pois no começo as ideias podem estar difusas, pobremente estruturadas ou ambíguas. A partir dos outros dois meios é produzida uma ata desorganizada.
- **Documentação:** requisitos descritos em itens diretos. Emprega língua natural e divide os requerimentos em categorias:
  - Requisitos funcionais: o que deve ser feito quanto à informatização das regras de negócio.
  - Requisitos não-funcionais: propriedade que a aplicação deve ter e que não estão diretamente relacionados com as regras de negócio. Exemplo: segurança do sistema (tempo de login e senha), tempo de processamento, disponibilidade (como 24/7)
  - Requisitos inversos: o que não é para fazer, deixar explícito aquilo que o cliente deixou implícito.
- **Verificação:** a equipe técnica verifica se o que está descrito na documentação é visível de ser desenvolvido. Exemplo: um dado requisito que não é possível de ser implementado.
- **Validação:** o cliente valida o projeto.

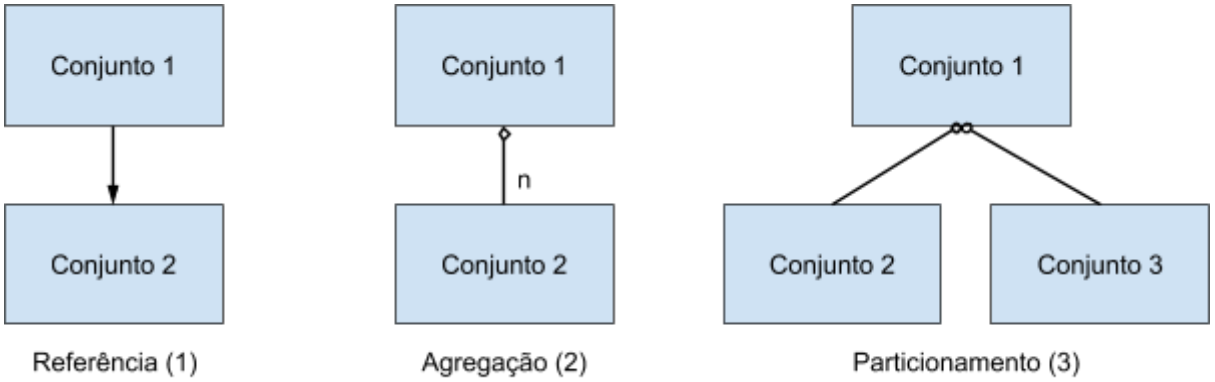
★ **Exemplos de Requisitos:**

- **Bem formulados:** a tela de resposta da consulta de aluno apresenta nome e matrícula; todas as consultas devem retornar respostas no máximo em dois segundos.
- **Mal formulados:** o sistema é de fácil utilização; a consulta deverá retornar uma resposta em tempo reduzido; a tela mostra seus dados mais importantes.

Aula 9

Modelagem de Dados

Um modelo deve ser base para n exemplos. Ou seja, um modelo mal estruturado é aquele para o qual podem-se listar ao menos um contra-exemplo que satisfaça às assertivas estruturais e ao desenho, mas que não condizem com o modelo. A notação utilizada para os módulos e relações entre eles é a seguinte:



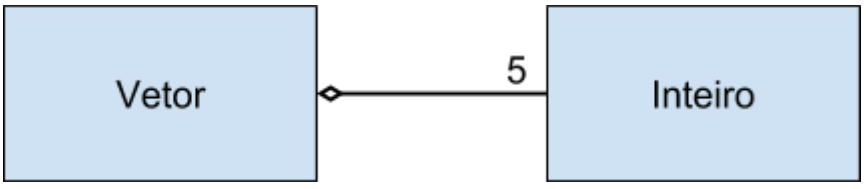
(1) Ponteiro (→) de um elemento referente a outro conjunto. Representa uma estrutura dinâmica, como uma lista.

(2) Diversos elementos de um conjunto fazem parte de outro. Representa uma estrutura estática, como um vetor.

(3) Os conjuntos 2 e 3 são especializações do conjunto 1, isto é, herdam características do conjunto 1, mas possuem alguma diferenciação.

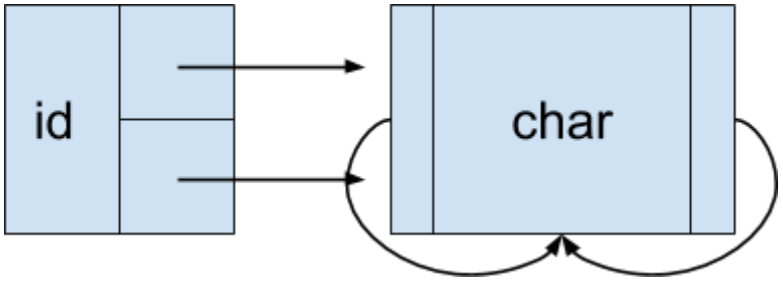
Exemplos:

a) **Vetor** de 5 posições de inteiros



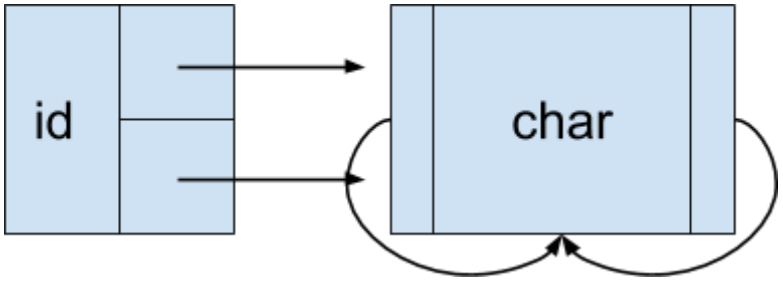
O conjunto vetor possui cinco elementos do tipo inteiro, como é indicado na relação de agregação com a seta com extremidade com losango.

b) **Árvore Binária** com cabeça que armazena caracteres



O conjunto Cabeça possui um identificador e dois ponteiros que apontam para o conjunto Nó. O conjunto nó possui uma variável de caractere e dois ponteiros que apontam para elementos deste mesmo conjunto, Nó.

c) **Lista duplamente encadeada** com cabeça que armazena caracteres



O modelo da lista encadeada com cabeça cujo nó armazena um caractere é idêntico ao da Árvore Binária com cabeça que armazena caracteres. Mas listas não são árvores binárias, apesar de apresentarem o mesmo desenho. O modo de diferenciá-las é por meio das assertivas estruturais, que são regras utilizadas para desempatar modelos iguais. Essas regras complementam o modelo definindo características que o desenho não conseguiria representar. Veja como exprimimos as assertivas estruturais de Lista e Árvore Binária:

Lista

Assertivas:

Se pcorr->pant != nulo, então pcorr->pant->pprox = pcorr.

Se pcorr->prox != nulo, então pcorr->pprox->pant = pcorr

Árvore Binária

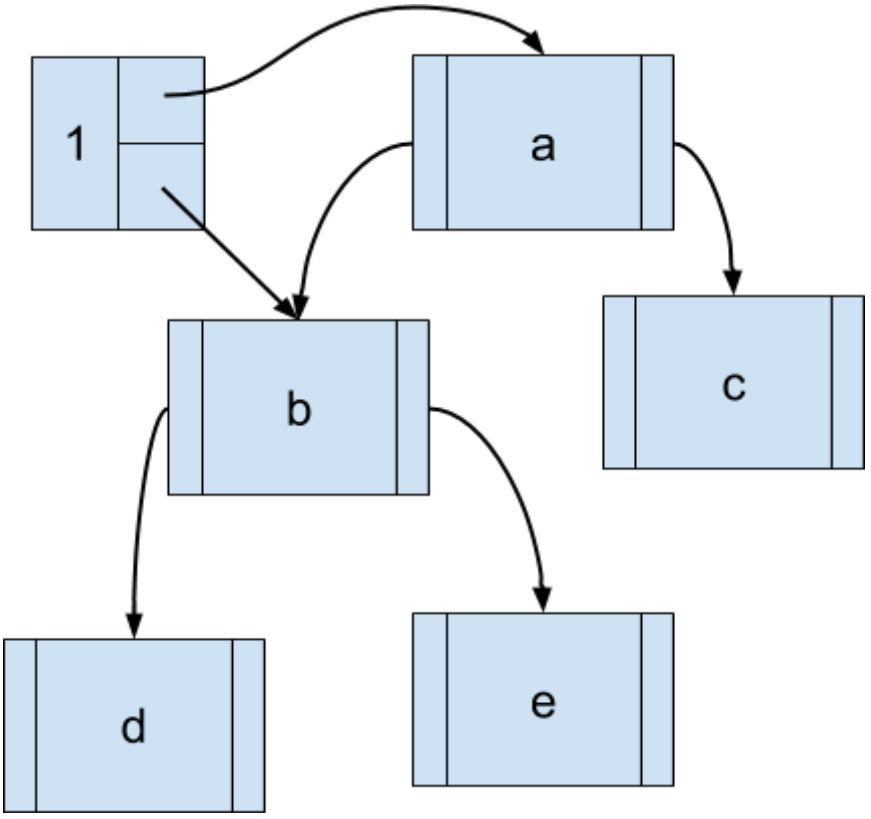
Assertivas:

Um ponteiro de uma subárvore a esquerda nunca referencia um nó de uma subárvore a direita.

Os ponteiros pant e pprox de um nó nunca

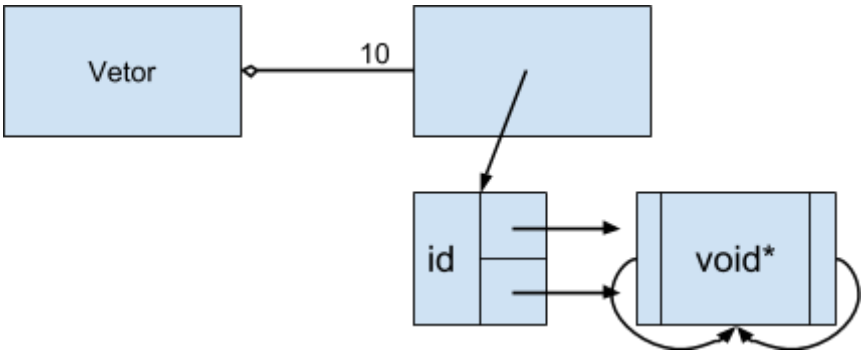
OBS: Nos trabalhos 2 e 3 do curso, deverão ser entregues os modelos das estruturas que estão sendo criadas nos TADs, assertivas para elas, e exemplos.

Exemplo de Árvore Binária:



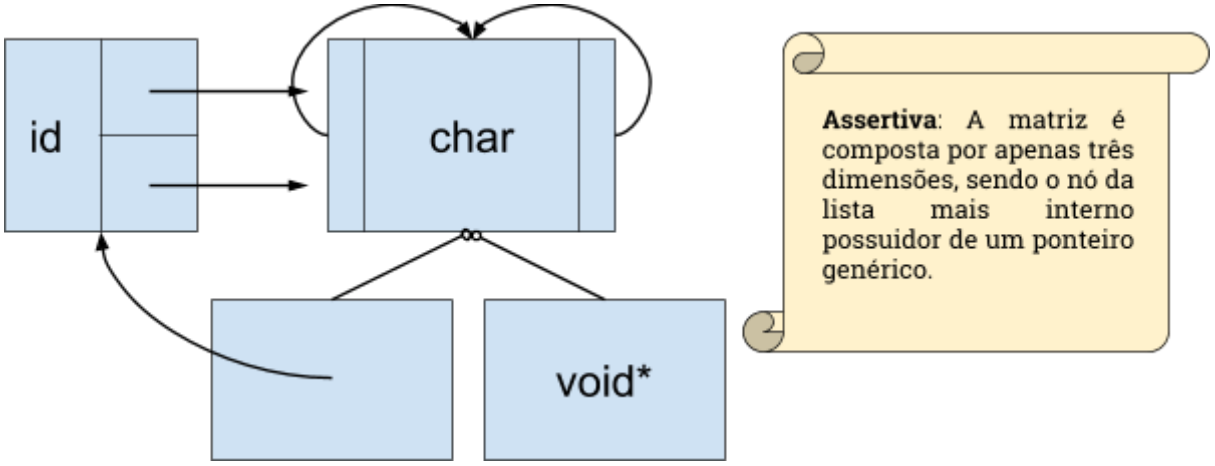
Perceba que no exemplo, os tipos das variáveis são substituídos por valores arbitrários, e ponteiros apontam para outros elementos, e não para si mesmos, necessariamente.

d) **Vetor de listas duplamente encadeadas** com cabeças e **genéricas**



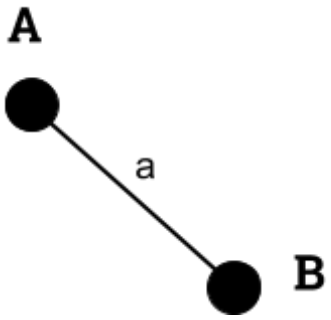
A variável do tipo (void *) é um ponteiro genérico, isto é, guarda o endereço de uma variável de qualquer tipo.

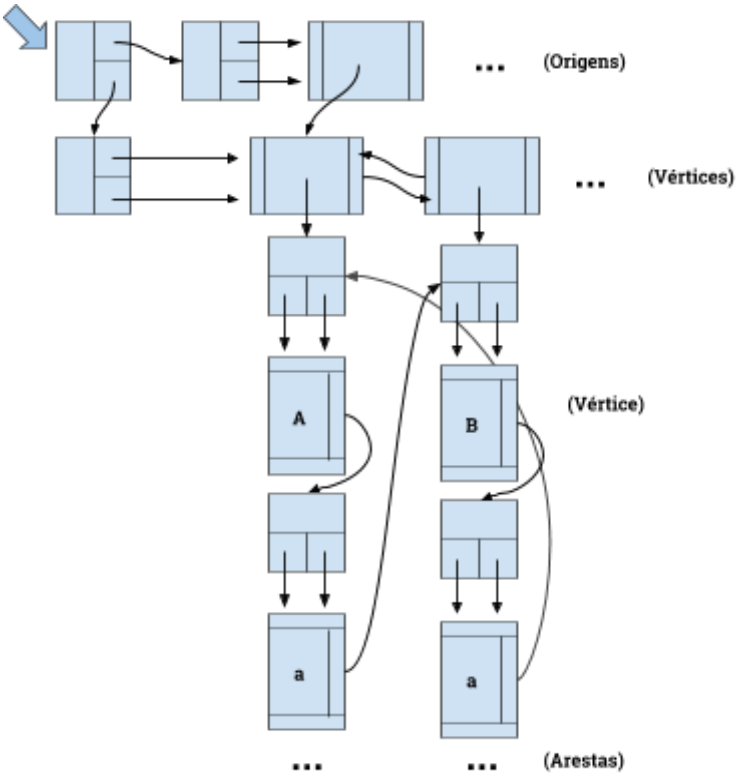
e) **Matriz tridimensional genérica construída com listas duplamente encadeadas** com cabeça



f) Grafo criado com listas duplamente encadeadas

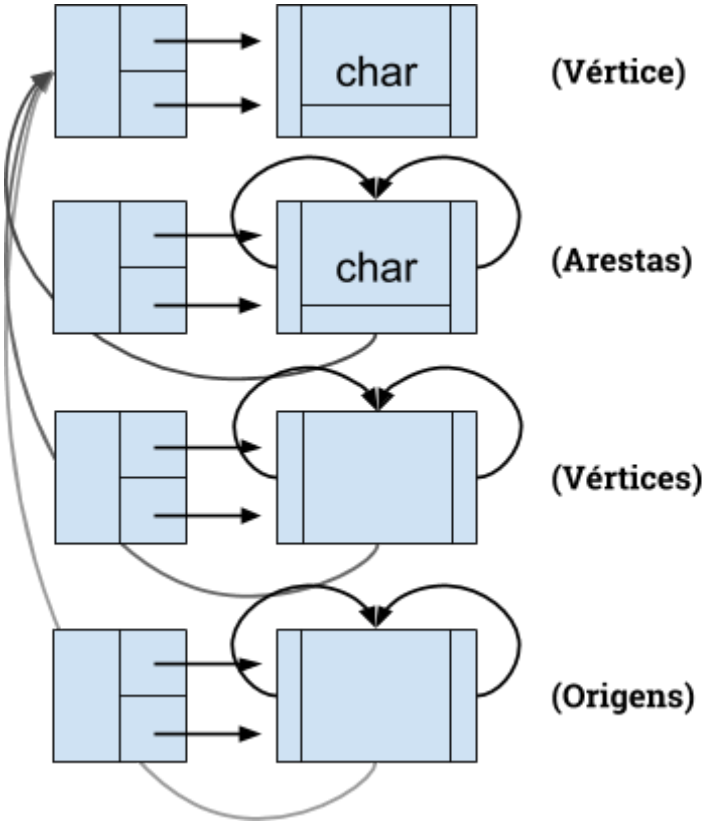
Abaixo, a representação em grafo, e, ao lado, o modelo deste mesmo grafo em listas.





A seta azul indica a Cabeça para o grafo em si. Este aponta para uma cabeça que trata das origens (vértices de partes desconexas do grafo), e outra cabeça que trata dos vértices. A lista de vértices contém ponteiros para cabeças de uma listas com um só Nó, que contém o nome do Vértice e um ponteiro para a cabeça da lista de arestas. Estas possuem ponteiros para o Vértice do outro lado da aresta.

Modelando, obtemos o seguinte desenho:



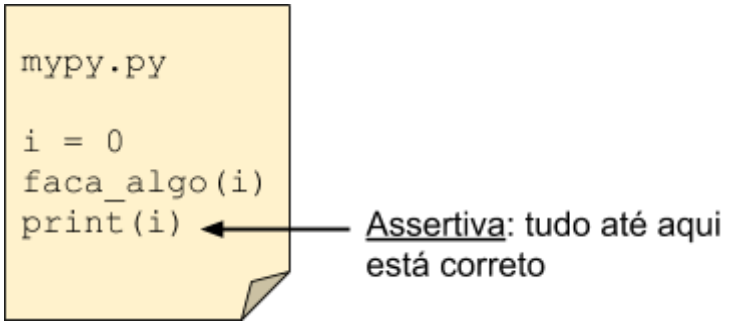
Aula 10

Assertivas

★ **Definição:**

○ Qualidade por construção: realizar com qualidade aplicada cada etapa de desenvolvimento de uma aplicação.

○ Assertivas são regras consideradas válidas em determinado ponto do código.



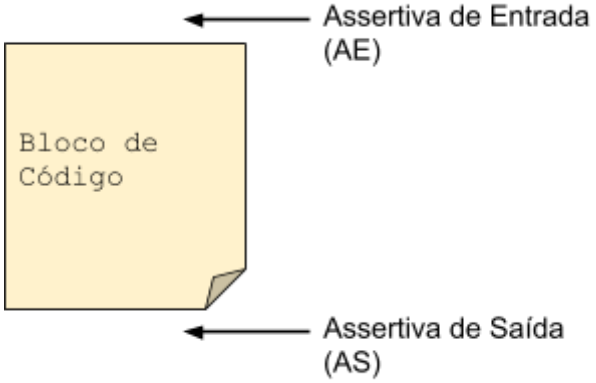
★ **Onde aplicar assertivas:**

○ Argumentação de corretude: provar que aquele bloco de código está correto.

○ Instrumentação: transforma assertivas em blocos de código (trabalho 4).

○ Trechos complexos em que é grande o risco de erros.

★ **Assertivas de Entrada e Saída:**



○ Obs: a assertiva de tratar de regras envolvendo dados e ações tomadas em um trecho do código. Exemplo: em AE o nó será inserido, e em AS o nó foi inserido.

★ **Exemplos:**

○ AE: a lista existe; o ponteiro corrente referencia o nó a ser excluído, que é intermediário.

○ AS: nó inicialmente referenciado pelo ponteiro para o nó corrente foi excluído; corrente aponta para o nó anterior.

Aula 11

Implementação da Prog. Modular

★ **Espaço de Dados**: São áreas de armazenamento, e...
- Alocadas em um meio
- Possuem um tamanho
- Possuem um ou mais nomes de referência

A[j] é o j-ésimo elemento do vetor A.
ptAux* é o espaço de dados apontado / referenciado por ptAux.
(*ObterElemento(int id)).Id é o subcampo do espaço de dados retornado pela função, que também pode ser obtido pela sintaxe ObterElemento(int id)→Id.

★ **Tipos de dados:** determinam a 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.

★ **Tipo de tipo de dados:**
- Tipos computacionais: int, char, char*...
- Tipos básicos (para desenvolvedor): enum, typedef, struct[1], union…
- Tipos abstratos de dados (TADs)

OBS: Conversão de tipos não é igual a imposição de tipos (typecast). Conversão é converter “1” (string) para 1 (inteiro). Imposição não altera o binário (como de signed int para unsigned int).

★ **Tipos Básicos**
- **Typedef:** dá um apelido a outro tipo
- **Enum:** enumera apelidos idênticos a um inteiro em ordem
- **Struct:** dados conglomerados
- **Union:** é um typecast mais organizado, pois interpreta o campo da estrutura com tipos diferentes. Exemplo: armazena como inteiro e reinterpreta como long int, ou então armazena como inteiro e reinterpreta como char.

★ **Declaração e definição de elementos:**
- **Definir:** alocar espaço e atrelar este espaço a um nome (binding)
- **Declarar:** associa um espaço de dados a um tipo

OBS: quando o tipo é computacional, ocorre simultaneamente a declaração e a definição daquela estrutura de dados.

[1] Tipos personalizados

★ **Implementação em C/C++:**

- Declarações e definições de nomes globais **exportados** pelo módulo servidor

```
  int a;
int F(int b);
```

- Declarações **externas** contidas no módulo cliente e que somente declaram o nome sem ser associado a um espaço de dados.

```
  extern int a;
extern int F(int b);
```

Deve existir ao menos um módulo com a real variável “a”. Ela é compartilhada por todos os módulos clientes. Esta linhagem é feita apenas em tempos de link-edição.

- Declarações e definições de nomes globais **encapsuladas** no módulo

```
  static int a;
static int F(int b);
```

Este encapsulamento protege as estruturas de dados de um módulo dos módulos clientes, aumentando a qualidade da modularização.

Mesmo que o módulo servidor tenha uma variável com mesmo nome, não poderá ser acessada por módulos clientes.

Aula 12

★ **Resolução de Nomes Externos:** um nome externo somente declarado em um determinado módulo necessariamente deve estar declarado e definido em outro módulo. A resolução:

- Associa os declarados aos declarados e definidos
- Ajusta endereços para os espaços de dados

★ **Pré-processamento:** tudo que é precedido do caractere # é interpretado pelo pré-processador, e gera o código fonte. Os comandos assimilados pelo pré-processador são os seguintes.

```
  #define nome valor
```

Substitui “nome” por “valor” (meramente em texto).

```
  #include <arquivo> {ou}
#include "arquivo"
```

Inclui todo o texto do arquivo (bibliotecas padrões da linguagem C ou headers…)

```
  #undef nome
```

A partir dessa linha, não haverá mais a substituição da string nome

```
  #if defined (nome) {ou}
#ifdef nome
      valorV
#else
      valorF
#endif
```

Definição condicional de uma string

```
  #if !defined (nome) {ou}
#ifndef nome
```

Negação da condição de definição de string

Exemplo 1:

```
  #if !defined (EXEMP_MOD)
#define EXEMP_MOD
      {texto do .h}
#endif
```

Evita que o módulo de definição seja duplicado, caso direta ou indiretamente seja incluído mais de uma vez pelo módulo cliente.

Exemplo 2:

M1.H

```
  #if defined (EXEMP_OWN)
#define EXEMP_EXT
#else
#define EXEMP_EXT extern
#endif

EXEMP_EXT int vetor[7]
#if defined EXEMP_OWN
      = {1,2,3,4,5,6,7};
#else
      ;
#endif
```

M1.C

```
  #define EXEMP_OWN
#include "M1.h"
#undef EXEMP_OWN
```

Produz a seguinte interpretação do módulo M1.H:

```
  int vetor[7] = {1,2,3,4,5,6,7};
```

M2.C

```
  #include "M1.H"
```

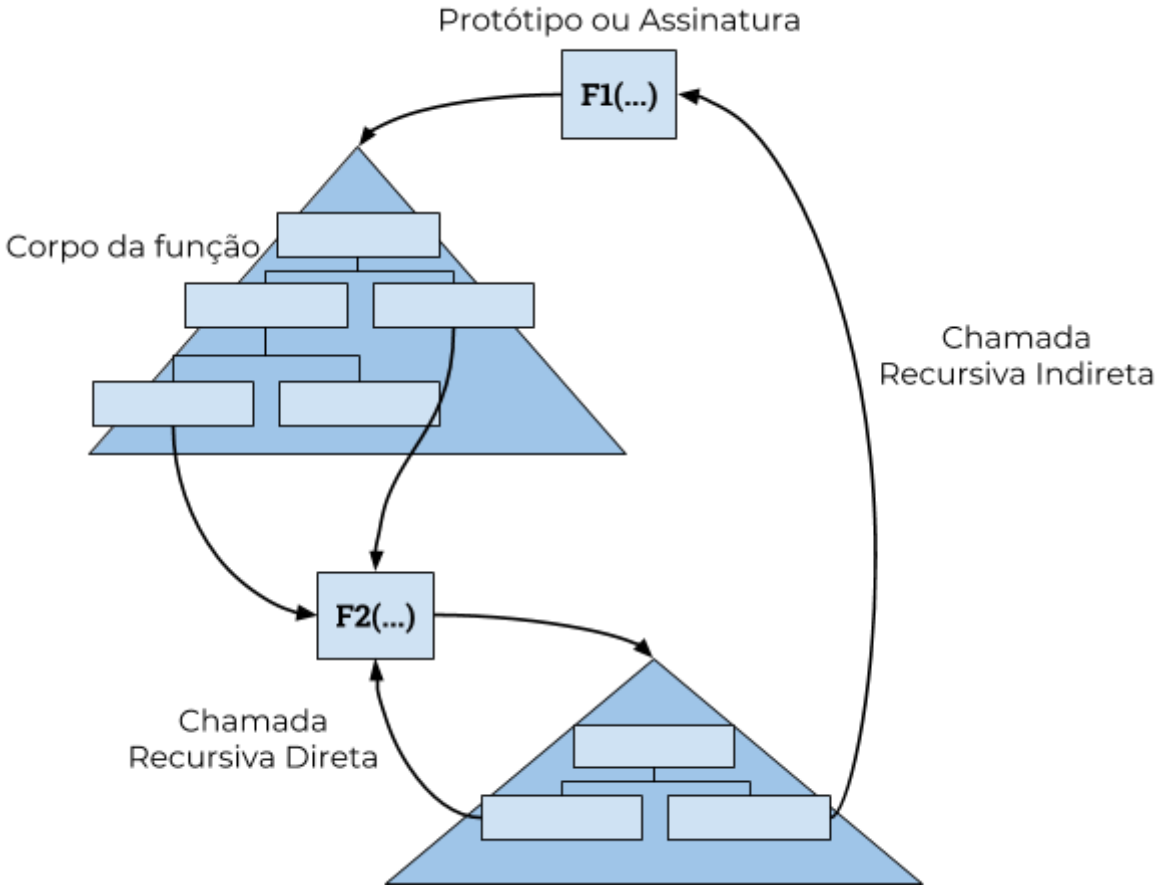
Produz outra interpretação do módulo M1.H:

```
  extern int vetor[7];
```

Aula 13

Estrutura de Funções

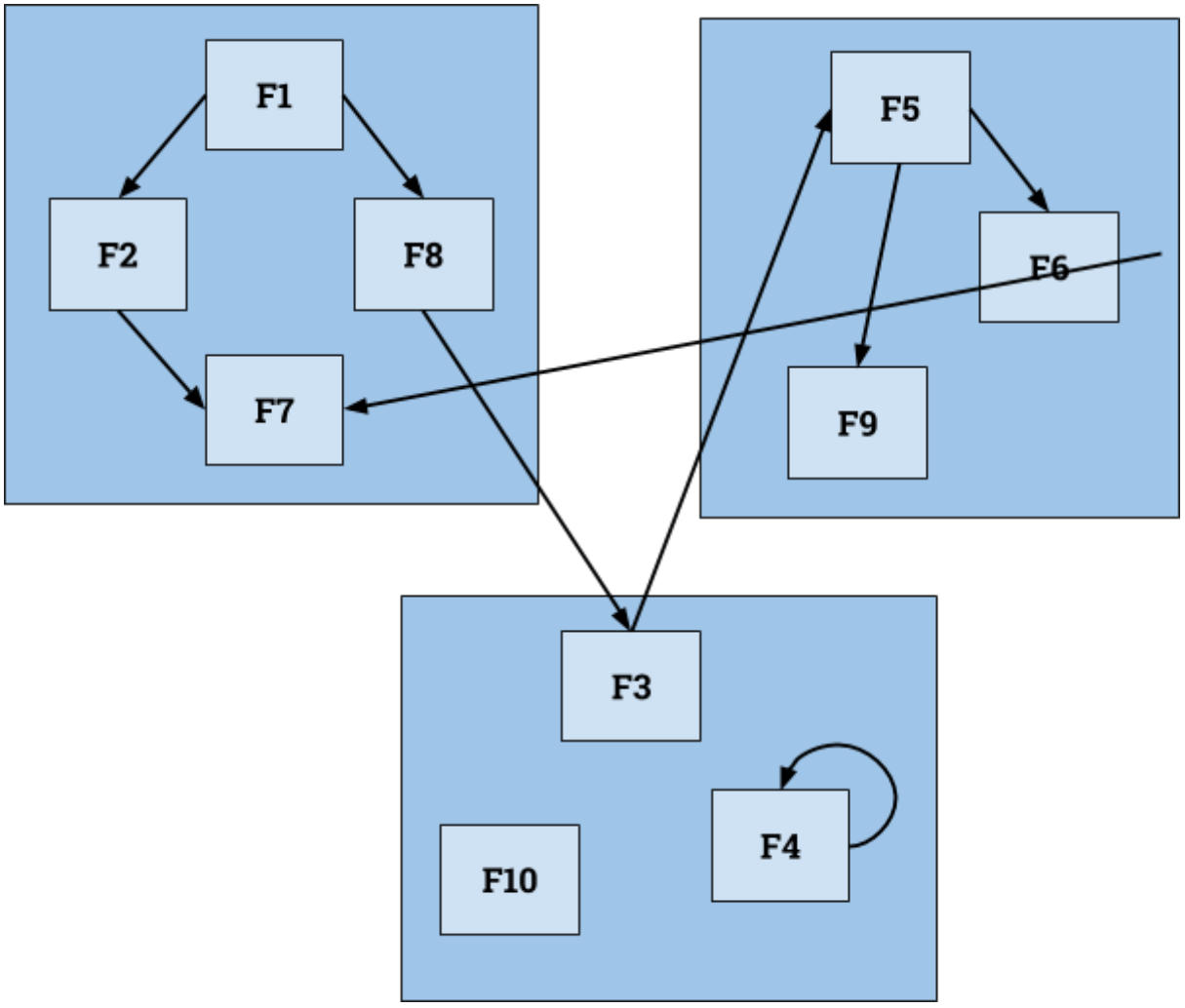
- ★ **Paradigma:** é a forma de programar
  - ○ Procedural: programação estruturada
  - ○ Orientada a Objetos:
    - ■ Programação Orientada a Objetos
    - ■ Programação Modular (mesmo utilizando uma linguagem não-orientada a objetos)
- ★ **Estrutura de Função:**



A estrutura interna das funções é ilustrada por uma árvore binária em que cada nó folha simboliza uma saída diferente da função para determinadas entradas, chamando ou não outras funções para gerar o retorno devido.

Chamamos de **arco de chamada** um certo caminho de chamadas de funções, que pode ser ou não **recursivo**, isto é, uma chamada que internamente faz outras chamadas da mesma função. A chamada recursiva pode ser **direta** (ou seja, dentro da função F2, chama-se F2), ou **indireta** (ou seja, dentro da função F2, chama-se F1, que chama F2).

★ **Estrutura de Chamadas:**




```
  F10
F4 -> F4
F9 -> F8 -> F3 -> F5 -> F9
F8 -> F3 -> F5 -> F6 -> F7
```

Chamamos **F10** de uma **função morta**, pois nesta aplicação ela não chama nem é chamada por nenhuma outra função. Vale ressaltar que uma função morta não é inútil, pois em outra aplicação, pode ser chamada.

A função **F4** faz uma **chamada recursiva direta**.

A função **F9** faz uma **chamada recursiva indireta**.

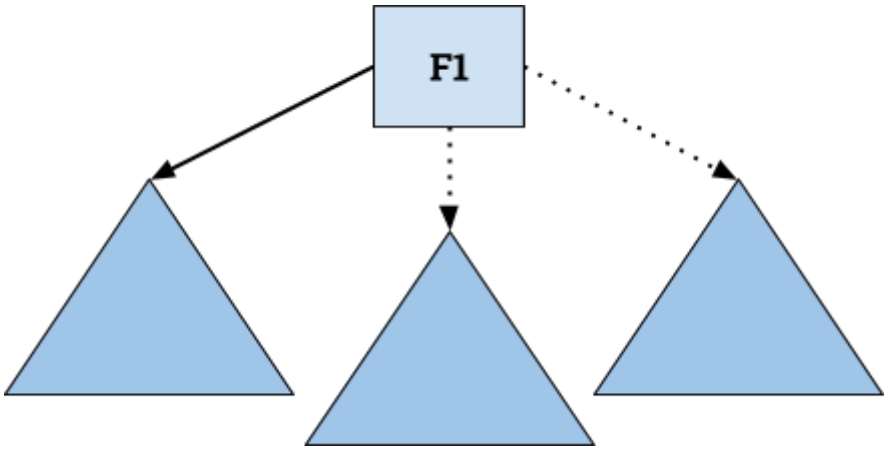
Já a última chamada exemplifica uma simples **dependência circular entre módulos.**

★ **Função:** porção autocontida de código, isto é, possui tudo o que é necessária para realizar uma tarefa. Possui:

- Nome
- Assinatura

- Um ou mais corpos de código

OBS: podemos ter ponteiros para funções, que variam de corpo de função. No exemplo a seguir vemos que **F1** é um ponteiro de função, que pode apontar para qualquer um dos três corpos de função.



★ **Especificação da Função:**

- **Objetivo:** o que a função deve fazer, que pode ser igual ao nome.
- **Acoplamento:** parâmetros e condição de retorno
- **Condições de acoplamento:** assertivas de entrada e assertivas de saída (os dados antes e depois a função ser executada)
- **Interface com usuário:** mensagem disponibilizada para usuário final da aplicação, saídas em tela, etc…
- **Requisitos:** o que deve ser feito pela função (como se fossem as assertivas intermediárias, no sentido do detalhamento do funcionamento interno, e não só entrada e saída da função)
- **Hipótese:** são regras pré-definidas que assumem como válidas uma determinada ação ocorrendo fora do escopo, evitando assim o desenvolvimento de código desnecessário.
- **Restrições:** são regras que limitam a escolha das alternativas de desenvolvimento para uma determinada solução. (limitado por tempo, capacidade de máquina, etc)

★ **Interfaces:**

- **Conceitual:** definição da interface da função sem preocupação com a implementação.

```
inserirSimbolo( Tabela , Símbolo ) -> Tabela, IdSimb, condRet
```

- **Física:** implementação da interface conceitual, com a sintaxe apropriada de uma linguagem de programação.

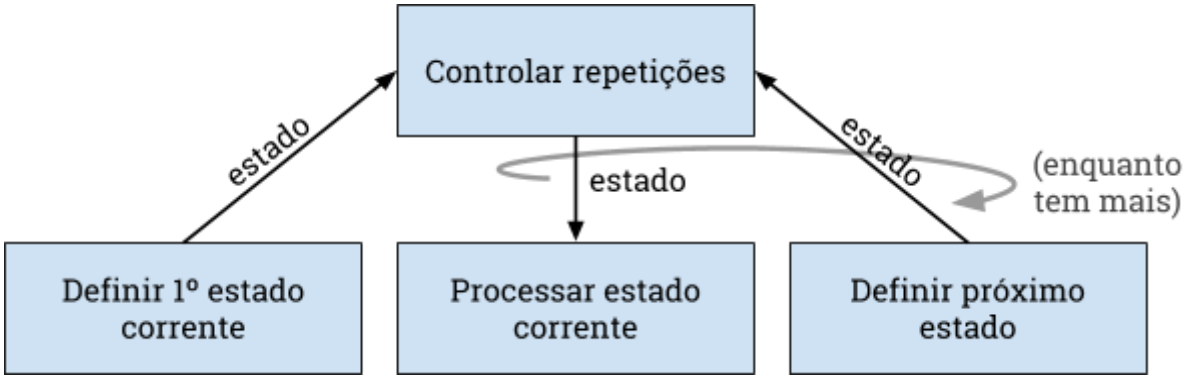
```
tpCondRet insSimb ( tpSimb * simbolo )
//Tabela -> global static no módulo
```

- **Implícita:** dados da interface diferentes de parâmetros e valores de retorno, como uma variável global que é padronizada, e é usada na função. Por exemplo, o ponteiro global para árvore no módulo ÁRVORE.

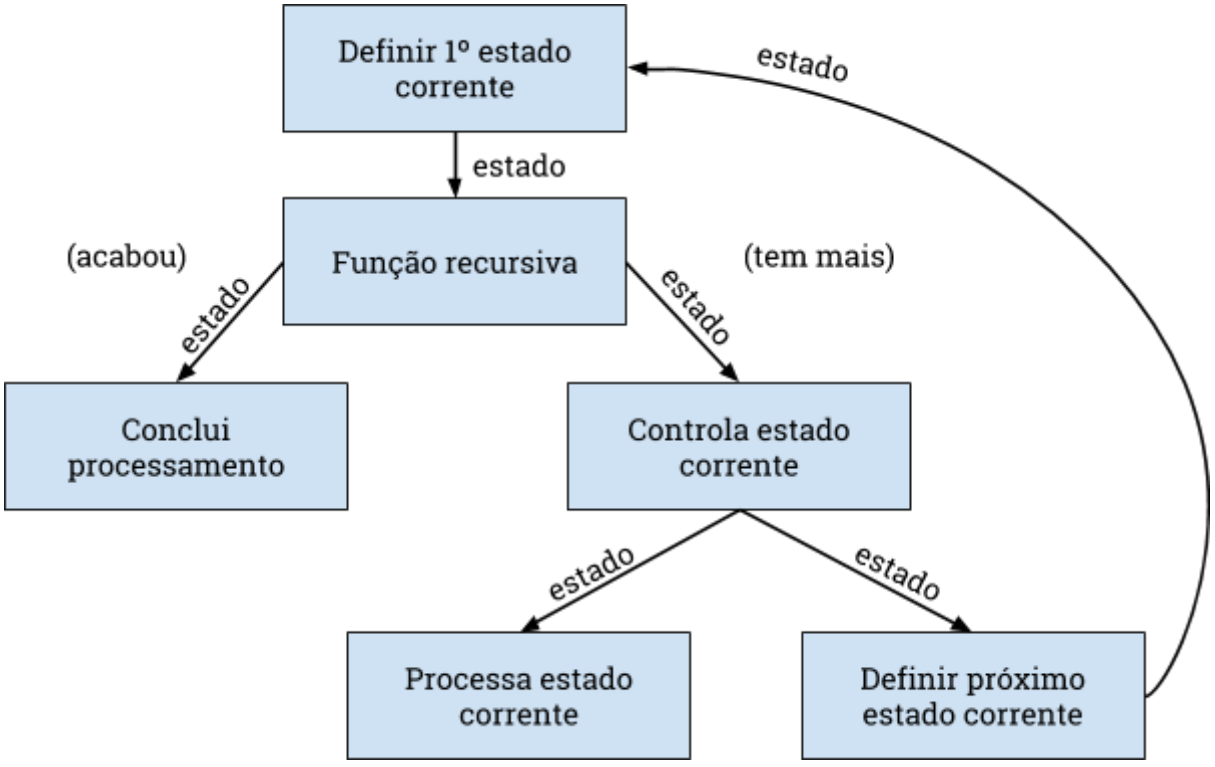
Aula 14

★ **Housekeeping:** código responsável por liberar componentes e recursos alocados dinamicamente a programas ou funções ao terminar sua execução.

★ **Repetições:**



★ **Recursão:**



★ **Estado:**

- **Descritor de estado:** conjunto de dados que definem um estado. Exemplo: índice de um vetor, limites de uma busca binária.
- **Estado:** é a valoração do descritor.
- **OBS:** Não é necessariamente observável. Exemplo: cursor de posicionamento de arquivo (fread, fwrite).

- **OBS:** Não precisa ser único, isto é, pode ser composto. Exemplo: limites inferior e superior de uma busca binária.

★ **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; }

 }
```

Funções como ObterLimSup, ObterLimInf e ObterValor constituem a **parte específica** de um programa, também conhecido como **hotspot**. Já trechos de código que não dependem da parte específica do código constituem a **parte genérica**. Ambos somados formam a **framework**. Esquemas de algoritmos permitem encapsular a estrutura de dados utilizada. É completo - independende de estrutura - e é incompleto - precisa ser instanciado.

Normalmente ocorre em:
- ➢ Programação orientada a objetos
- ➢ Frameworks

OBS: Se o esquema estiver correto e o hotspot estiver com assertivas válidas, então o programa está 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; }

 int processaArea( float valor1, float valor2, float (*Func)(float,float)
 )
 { printf("%f", Func(valor1,valor2)); }

 CondRet = processaArea(5,2,areaQuad);
 CondRet = processaArea(3,3areaTrig);
```

Aula 15

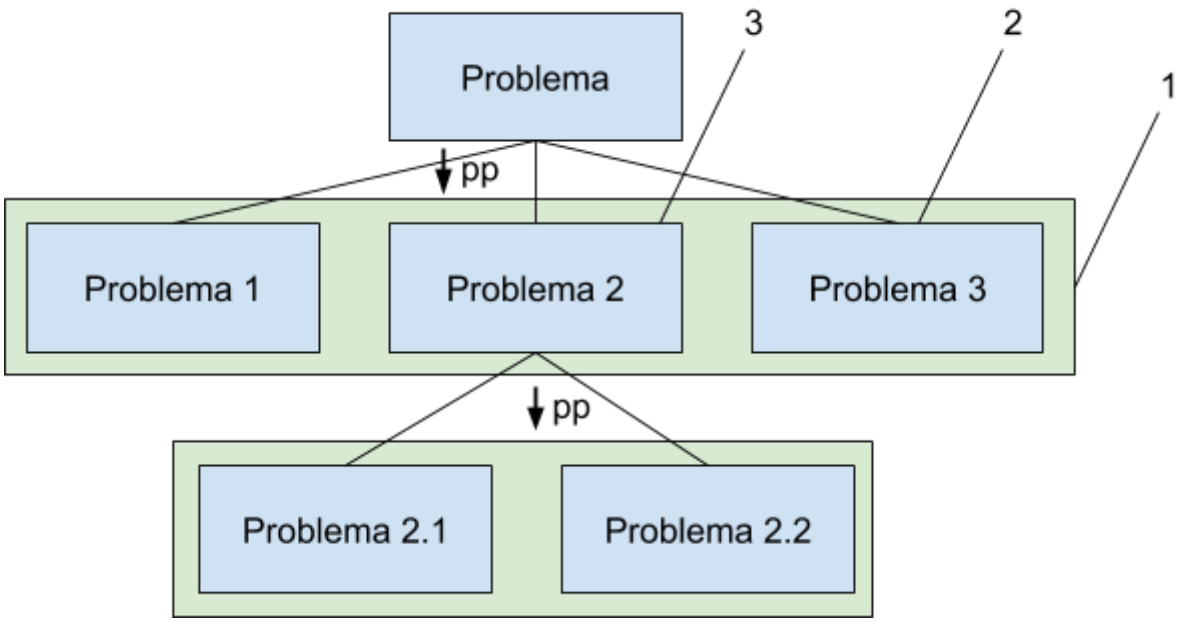
Decomposição Sucessiva

★ **Conceito:**

○ **Divisão e conquista:** dividir um problema em subproblemas menores de forma que seja possível resolvê-los.

○ A decomposição sucessiva é um método de divisão e conquista.

★ **Estruturas de Decomposição:**



1) **CONJUNTO SOLUÇÃO:** todos os componentes que englobam o mesmo problema
2) **COMPONENTE ABSTRATO:** aquele que se subidive em outros componentes
3) **COMPONENTE CONCRETO:** aquele problema/componente que não precisa ser subdividido em problemas menores.

○ Problema 2.1 e problema 2.2 formam o conjunto solução do componente problema 2. São os sub-componentes que precisam ser solucionados para que o componente-pai seja considerado solucionado.

○ No diagrama anterior, o Problema é o componente raíz.

○ Uma estrutura de decomposição gera uma solução apenas.

○ Uma solução pode ser destrinchada por múltiplas estruturas de decomposição diferentes (boas e ruins).

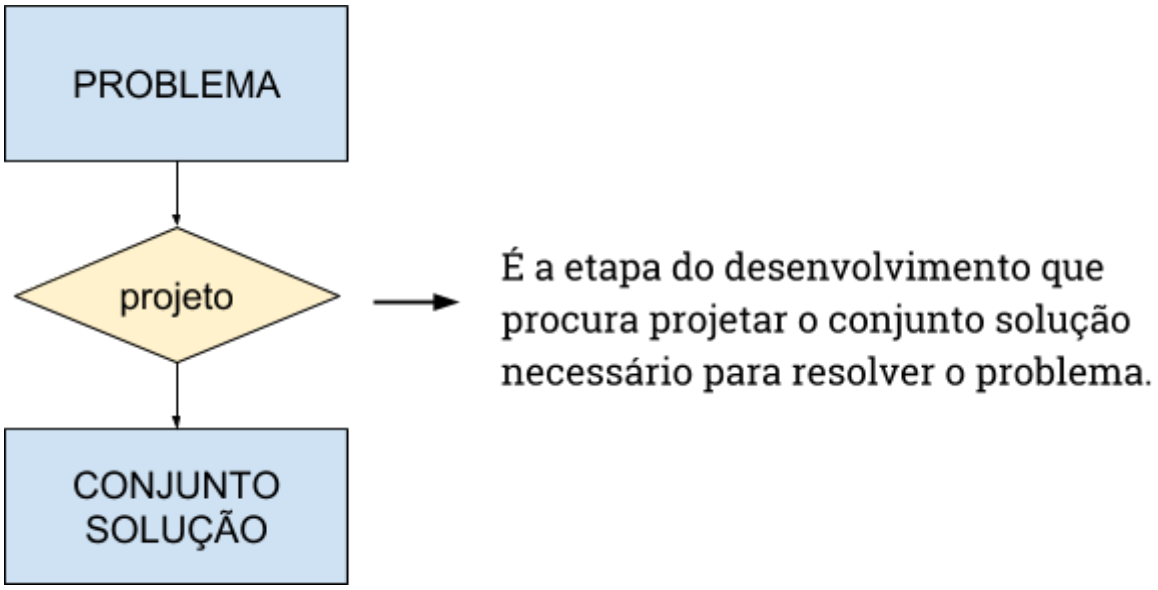
★ **Critérios de Qualidade:**

○ **Complexidade:** quantidade exagerada de etapas/sub-componentes

○ **Necessidade:** sub-componente fora do escopo, que não altera a solução do problema. (Acontece geralmente no controle de versão, quando há uma mudança na estrutura, mas as funções de acesso não são atualizadas)

- **Suficiência:** os componentes que fazem parte do conjunto solução são suficientes para resolver o problema? Não faltam sub-componentes?
- **Ortogonalidade:** dois componentes não realizam a mesma funcionalidade, disjuntos, dentro do mesmo conjunto solução.
- **OBS:** se uma mesma funcionalidade repetir em uma solução, provavelmente trata-se de uma função, que é chamada em mais de um componente.
- **OBS:** Um componente pode ser sub-componente de mais de um componente. (Exemplo: uma função que verifica um dígito identificador de diversas estruturas, que são implementadas em componentes em nível acima na estrutura de decomposição)

★ **Passo de Projeto:**



★ **Direção de Projeto:**

- **Bottom-up:** Primeiro são implementados os sub-componentes mais internos, para então implementar os mais acima na estrutura de decomposição.
- **Top-down:** Primeiro são implementados os componentes principais para então desenvolver os componentes mais específicos. Neste sentido, as funções “fakes”, que retornam valores fantasiosos, são a solução temporária para permitir a implementação de programas mais acima, sem precisar implementar componentes abaixo.
- **OBS:** É possível conciliar as duas direções de projeto.

Aula 16

Argumentação de Corretude

```
 INICIO
  IND ← 1
  ENQUANTO IND <= LL FAÇA
     SE ELE[IND] = PESQUISADO
        BREAK
     FIM-SE
     IND ← IND + 1
  FIM-ENQUANTO
  SE IND <= LL
     MSG "ACHOU"
  SENÃO
     MSG "NÃO ACHOU"
  FIM-SE
FIM
```

★ **Definição:** é um método utilizado para argumentar que um bloco de código está correto (mais para fins acadêmicos, mas também empregado no mercado para trechos de códigos não-triviais).

★ **Tipos de argumentação:**
- **Sequência**: blocos um após o outro
- **Seleção**: if, switch, else
- **Repetição**: for, while

★ **Argumentação de sequência**:

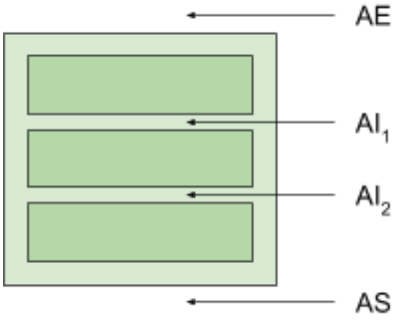
Assertivas intermediárias acontecem quando a assertiva de saída de um bloco coincide com a assertiva de entrada do bloco seguinte. No código-exemplo, as assertivas de entrada e saída e intermediárias do código seriam as seguintes:

AE: existe um vetor válido e um elemento a ser pesquisado.

AS: imprima “ACHOU” se o elemento foi encontrado ou “NÃO ACHOU” se end > LL[2].

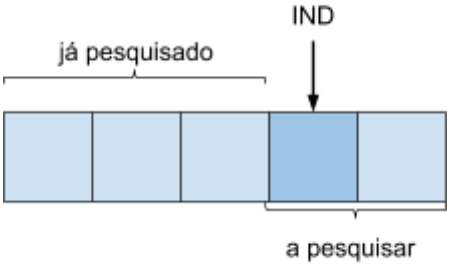
AI_1: IND aponta para primeira posição do vetor

AI_2: Se o elemento foi encontrado, IND aponta para o mesmo, senão IND > LL.



[2] Limite lógico, diferente de limite físico, que é o maior quantidade de informação que um espaço de dados pode armazenar. O limite lógico dita o fim de uma estrutura sequencial, como ‘\0’ na string.

★ **Argumentação de seleção**:



1) AE && (C == T) + B → AS
AE && (C == F) → AS

2) AE && (C == T) + B_1 → AS
AE && (C == F) + B_2 → AS

(1) Se as assertivas de entrada estiverem verdadeiras e a condição estiver verdadeira e o bloco **B** foi executado, a assertiva de saída é verdade.

(2) Pela AE, se o elemento for encontrado IND aponta o mesmo (e é menor do que LL) com (C == T), IND <= LL, **B_1** apresenta mensagem “ACHOU”, valendo AS.

Pela AE, IND > LL se o elemento pesquisado não foi encontrado. Como (C == F), INF > LL, neste caso **B_2** é executado apresentando mensagem “NÃO ACHOU”, valendo AS.

★ **Argumentação de repetição**:

AE: AI_1
AS: AI_2

AINV: assertiva invariante, envolve os dados descritores de estado, que deve ser válida a cada ciclo de repetição - isto é, não varia com os ciclos.

AINV: Existem dois conjuntos: já pesquisados e a pesquisar. INF aponta para elemento do a pesquisar.

1. **AE → AINV** : A assertiva invariante é válida antes da assertiva de entrada. Conjunto já pesquisado é vazio e IND aponta para primeiro elemento. Portanto todos os elementos estão no conjunto a pesquisar.
2. **AE && (C == F) → AS** : A condição será falsa se não entrar no loop, ou entrar mas não fechar o primeiro ciclo.

2.1. **Não entra**: pela AE = AI_1, IND = 1. Como (C == F), LL < 1. Ou seja, LL = 0. Ou seja, o vetor é vazio. Neste caso, é válido a assertiva de saída pois o elemento procurado não foi encontrado.

2.2. **Não completa o primeiro ciclo**: pela AE, IND aponta para o primeiro elemento do vetor. Se este for igual ao pesquisado, o break é executado e IND aponta para elemento encontrado, valendo AS.

3. **AE && (C == T) + B → AINV** : Pela assertiva, IND aponta para primeiro elemento do vetor. Como (C == T), este primeiro elemento é diferente do pesquisado. Este então passa do conjunto a pesquisar para os que já foram pesquisados e IND é reposicionado para outro elemento de a pesquisar, valendo AINV.
4. **AINV && (C == T) + B → AINV** : Para AINV continuar valendo, B deve garantir que um elemento passe de a pesquisar para já pesquisado e IND seja reposicionado.

5. **AINV && (C == F) → AS** : Ou a condição é falsa ou o ciclo não completa.

5.1. **Condição falsa:** pela AINV, IND ultrapassou o limite lógico e todos os elementos estão em já pesquisado. Pesquisado não foi encontrado com IND > LL, vale AS.

5.2. **Ciclo não completou:** pela AINV, IND aponta para elemento de a pesquisar que é igual a pesquisado. Neste caso vale a AS pois ELE[IND] = PESQUISADO.

6. **TÉRMINO** : como a cada ciclo, B garante que um elemento de a pesquisar passe para já pesquisado e o conjunto a pesquisar possui um número finito de elementos, a repetição termina em um número finito de passos.

Aula 17

Resta argumentarmos os blocos dentro do laço ENQUANTO. São dois blocos. Portanto, faremos a argumentação de sequência. Mas caso houvesse apenas um bloco de código, não seria necessário fazer uma argumentação de sequência.

★ **Argumentação de sequência (dentro do ENQUANTO)**:
AE: AINV, **AS**: AINV
AI_3: o elemento pesquisado não é igual a ELE[IND] ou o elemento foi encontrado em IND.

★ **Argumentação de seleção (dentro do ENQUANTO)**:
AE: AINV, **AS**: AI_3

1. **AE && (C == T) + B → AS** : Pela AE, IND aponta para elemento do conjunto a pesquisar. Como C == T, o elemento de IND = PESQUISADO e assim o elemento foi encontrado em IND, valendo a AS.
2. **AE && (C == F) → AS** : Pela AE, IND aponta para elemento a pesquisar. Como C == F, o elemento apontado por IND não é o pesquisado, valendo a AE.

Instrumentação

★ **Problemas ao realizar testes**
- **Esforço de Diagnose / Diagnóstico:**
  - Normalmente é grande
  - Muito sujeito a erros

  Contribui para este esforço…
  - Não estabelece com exatidão a causa a partir dos problemas observados
  - Tempo decorrido entre o instante da falha e o observado
  - Falhas intermitentes (que ora ocorrem ora não)
  - Causa externa ao código que mostra a falha
  - Ponteiro “louco” (*wild pointers*)
  - Comportamento inesperado do *hardware*

Aula 18

★ **O que é instrumentação?**
- Fragmentos inseridos nos módulos
  - Códigos
  - Dados (variáveis de controle)
- Não contribuem para o objetivo final do programa
- Monitora o serviço enquanto o mesmo é executado
- Consome recursos de execução (e, portanto, não deve ser ativada na versão a ser distribuída para o usuário)

★ **Objetivos**
- **Detectar falhas** de funcionamento do programa o mais cedo possível, de forma automática
- **Impedir** que falhas se propaguem
- **Medir propriedades dinâmicas** do programa (como tempo de execução, gargalos de um programa, memória etc)

★ **Conceitos**
- **Programa robusto** intercepta a execução quando observa um problema. Mantém o dado confinado, explanando ou sugerindo onde está o erro.
- **Programa tolerante a falhas** é robusto, e corrige o erro para integridade do programa. Possui mecanismos de recuperação.
- **Deterioração controlada** é a habilidade do programa de continuar operando corretamente mesmo com a parada de funcionalidade. Exemplo: quando o Microsoft Word permite que o usuário salve o arquivo mesmo quando o processo trava.

★ **Esquema de inclusão de instrumentação em C/C++**

```
#ifdef _DEBUG
      // código da instrumentação
      // dados
#endif
```

Este código deve rodar em tempo de desenvolvimento, na versão debug. E deve ser inativo na versão release.

★ **Assertivas executáveis**

ASSERTIVA → CÓDIGO

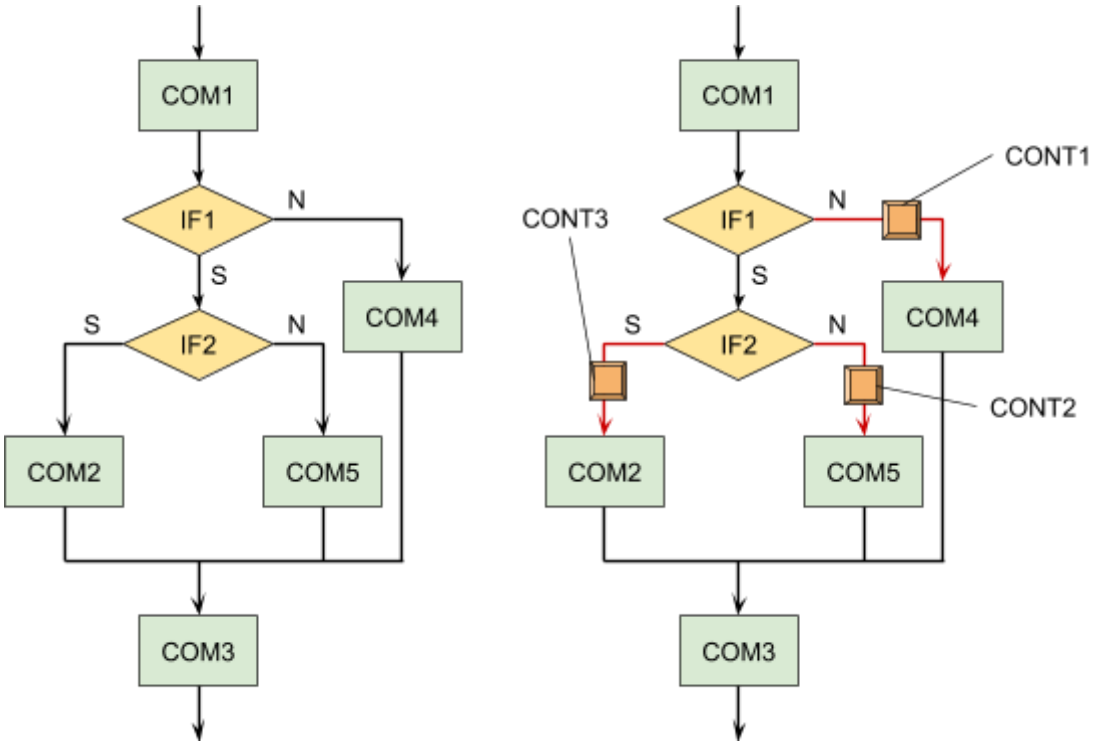
- **vantagens:**
  - Informam o problema quase imediatamente após ter sido gerado
  - Controle de integridade feito pela máquina
  - Reduz o risco de falha humana
  - Precisam ser: completas (cobrir todas as assertivas) e corretas

- ★ **Depuradores (Debuggers):** ferramenta utilizada para executar um código passo-a-passo, permitindo que se distribuem breakpoints, com o objetivo de confinar os erros a serem pesquisados. Deve ser utilizado como último recurso.
- ★ **Trace:** instrumento utilizado para apresentar uma mensagem no momento em que é executado (printf da linguagem C).
    - Trace de **instrução**: printf que apresenta mensagem.
    - Trace de **evolução**: apresenta mensagem apenas quando o conteúdo de uma variável for alterado, permitindo a visualização da evolução de um indicador de estado, como um índice de um vetor sendo analisado.

Aula 19

★ **Controlador de Cobertura**

- Teste **caixa fechada/preta**: observa-se apenas o que entra na função (parâmetros) e o que sai da função (condições de retorno, dados…)
- Teste **caixa aberta/branca**: analisa-se todos os caminhos internos da função.



Definição: instrumento composto de um vetor de controladores que tem como objetivo acompanhar os testes caixa aberta de uma aplicação, mostrando todos os caminhos percorridos. Caso um dos caminhos não foi percorrido (isto é, o contador neste caminho é zero), percebe-se que nem todos os caminhos possíveis foram verificados, e a instrumentação não é rigorosa o suficiente.

Vetor de controladores: armazena as labels e contador condizente com a quantidade de vezes que se passou por tal contador. Se pudéssemos modelar o fluxograma como uma árvore, os cortadores estariam nos nós-folha, pois pressupõem os “IF’s” passados.

LABEL	QUANTIDADE
CONT1	**7**
CONT2	**8**

CONT3	0 (!)

OBS: Em teste de caixa fechada, testa-se as condições de retorno; em caixa aberta, testam-se os caminhos.

★ **Verificador Estrutural:**

- Lembrando… se a assertiva de entrada, saída e intermediária que é implementada em código chama-se assertiva executável, agora faremos a mesma conversão de assertivas para código mas com as assertivas estruturais + modelo.

- **Definição:** instrumento responsável por realizar uma verificação completa da estrutura em questão é a implementação de código relacionado com as assertivas estruturais e modelo.

★ **Deturpador Estrutural:**

- **Definição:** Instrumento responsável por inserir erros na estrutura com objetivo de testar o verificador.

- A função **deturpa** recebe o **tipo de deturpação**. A deturpação é sempre realizada no nó corrente.

```
=criaestrutura
=deturpar 1
=verifica
=deturpar 2
=verifica
...
=estatisticacontador
```

★ **Recuperador Estrutural**

- **Definição:** instrumento responsável por recuperar automaticamente uma estrutura a partir de um erro observado em uma aplicação tolerante a falhas.

★ **Estrutura Autoverificável:**

- **Definição:** é estrutura que contém todos os dados necessários para que o programa esteja totalmente verificado. Para isso são feitas algumas perguntas à uma lista simplesmente encadeada genérica.

É possível definir todos os tipos de dados apontados pela estrutura?
Não, basta armazenarmos então o tipo apontado por estrutura. Este “tipo” será passado pela função no módulo específico na versão instrumentada (debug) do programa. Vale lembrar que na chamada e definição da função de inserção de nó, deve-se passar este tipo. Além disso, é necessário incluir o campo “tipo” à cabeça da lista, para que nós não tenham o mesmo tipo de cabeça.

É possível acessar qualquer parte da estrutura a partir de qualquer origem?
Não, basta então armazenar um ponteiro para a cabeça da lista e um ponteiro para o nó anterior, tornando a lista duplamente encadeada.

```
Condret inserirno( pt, valor
      #ifdef _DEBUG
            , tipo
      #endif
            );
```