



# LABORATÓRIOS DE INFORMÁTICA III

**Fernando Mário Martins**

**2016/2017**

**MIEI: 2º Ano, 2º Sem**

**Vítor Francisco Fonte** ([vff@di.uminho.pt](mailto:vff@di.uminho.pt))

**Francisco Maia** ([fmaia@di.uminho.pt](mailto:fmaia@di.uminho.pt))

**João Paulo** ([jtpaulo@di.uminho.pt](mailto:jtpaulo@di.uminho.pt))

**DI/UM**

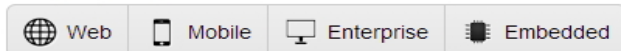
- ▣ **Conhecer os princípios fundamentais da Engenharia de Software**, designadamente **modularidade, reutilização, encapsulamento e abstracção de dados**, e saber implementá-los em diferentes linguagens/paradigmas de programação: (imperativo em **C** - 1º projecto e POO em **Java** - 2º projecto);
- ▣ **Complementar experimentalmente os conhecimentos adquiridos** nas Unidades Curriculares de Programação Imperativa, Algoritmos e Complexidade, Arquitectura de Computadores e Programação Orientada aos Objetos;
- ▣ **Desenhar (conceber), codificar e testar software**, realizando dois projectos concretos de média dimensão,
  - **1º projeto - Linguagem C**: modularidade, reutilização, encapsulamento, estruturas de dados, manipulação de ficheiros, etc.;
  - **2º projeto - Linguagem Java**: classes, packages, herança, reutilização de código, polimorfismo, colecções e streams de I/O;





















## Linguagens em 2015 (sem modas nem tiques)

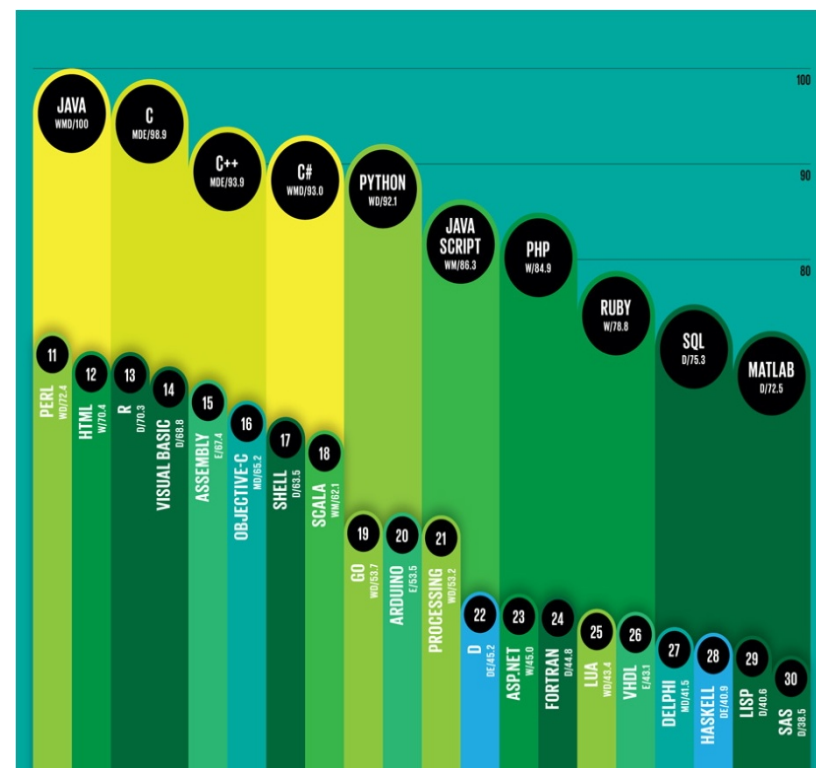
Atenção desenvolvedores! De acordo com o ranking da IEEE, a linguagem de programação Java é a que vem sendo mais utilizada no mundo. A pesquisa foi feita a partir de 12 fontes de dados, incluindo Google, GitHub, Stack Overflow e o fórum Hacker News.

Ao todo, 49 linguagens de programação entraram na lista e você pode conferir o ranking completo através do [site do IEEE](#). O Java aparece em primeiro lugar no ranking, atingindo 100 pontos, mas é seguido de perto pela linguagem C com 99,2 pontos e em terceiro lugar está o C++ com 95,5. Outras linguagens de programação web também muito utilizadas, PHP e Ruby, só aparecem na sexta e oitava posições, respectivamente. Já a a linguagem de programação Swift, da Apple, não aparece no ranking, mas sua antecessora, Objective-C, está na décima sexta posição.

### Language Types



Language Rank	Types	Spectrum Ranking
1. Java	  	100.0
2. C	  	99.2
3. C++	  	95.5
4. Python	 	93.4
5. C#	  	92.2
6. PHP		84.6
7. Javascript	 	84.3
8. Ruby		78.6
9. R		74.0
10. MATLAB		72.6



▣ **As PLs são momentos reservados a apoio tutorial aos alunos** que necessitem de esclarecer dúvidas e/ou precisem de acompanhamento para a execução dos projectos. **A frequência é obrigatória 2 horas/semana.**

► **Os alunos realizarão dois projectos práticos obrigatórios.**

- **O 1º projecto de C será de dimensão média e realizado em grupo (máx. 3 alunos) e terá apenas a submissão final e avaliação presencial.**

- **O 2º projecto, de JAVA, será realizado em grupo (máx. 3 alunos) e terá apenas a submissão final e avaliação presencial.**

► **A fórmula que calcula a nota final pressupõe que ambos os trabalhos foram entregues e ambos possuem avaliação final  $\geq 10$ :**

$$\text{Nota Final} = 60\% \cdot \text{ProjC} + 40\% \cdot \text{ProjJava}$$

---

### TURNOS DISPONÍVEIS (inscrições são desnecessárias):

3<sup>a</sup>. Feira, 18H00-20H00 (PL1 - CP1.314)

5<sup>a</sup>. Feira, 14H00-16H00 (PL3 - DI.0.05)

5<sup>a</sup>. Feira, 14H00-16H00 (PL5 - DI.0.03)

5<sup>a</sup>. Feira, 16H00-18H00 (PL4 - DI.0.11)

6<sup>a</sup>. Feira, 17H00-19H00 (PL2 - DI.0.05)

- Inscrições nos grupos práticos a realizar no BB (vão ser criados os 80 grupos)

## **1º projecto (C):**

**23 de Abril: data limite de entrega (23:59)**

**26-28 de Abril: apresentação e discussão**

## **2º projecto (Java):**

**4: data limite de entrega (23:59)**

**7-9 de Junho: apresentação e discussão**

---

## 2015-2016

Semana	2a.feira	3a.feira	4a. Feira	5a.feira	6a.feira	sábado	
8/02 a 13/02							Semana de MIEI
15/02 a 20/02			COMUM				Aula comum de apresentação de LI3 (TP de C)
22/02 a 27/02							
29/02 a 5/03							
7/03 a 12/03							
14/03 a 19/03							
21/03 a 26/03							Páscoa
28/03 a 2/04							
4/04 a 9/04							
11/04 a 16/04			COMUM				TP de Java + Entrega electrónica do TP de C
18/04 a 23/04							Avaliações do TP de C
25/04 a 30/04							
2/05 a 7/05							
9/05 a 14/05							Semana da queima
16/05 a 21/05							
23/05 a 28/05							
30/05 a 4/06							
6/06 a 11/06							Entrega electrónica do TP de Java
13/06 a 18/06							Avaliações do TP de Java
20/06 a 25/06							Lançamento das Notas Finais
27/06 a 2/07							

Aula comum em sala a marcar
 entregas electrónicas de trabalhos
 avaliações presenciais dos trabalhos
 laboratórios de C
 laboratórios de Java
 férias feriados
 queima
 Notas finais da UC

■ EM INFORMÁTICA, E QUALQUER QUE SEJA A PERSPECTIVA, HÁ APENAS DOIS TIPOS DE ENTIDADES COMPUTACIONAIS:

▣ INFORMAÇÕES;

▣ TRANSFORMADORES DE INFORMAÇÕES;

■ COMO SÃO CARACTERIZADAS ?

● PELA FORMA ► SINTAXE

● PELO SIGNIFICADO ► SEMÂNTICA

Passamos a vida a estudar sintaxe e semântica (isto é, **linguagens**)

**PARADIGMA = MODELO COMPUTACIONAL**

Um **modelo computacional** é uma abstracção (simplificação) do processo computacional concreto que se realiza na máquina, que nos permite racionalizar de uma forma simples como é que **informações** e **transformadores** interagem para realizar a **computação**.



## PARADIGMAS TRADICIONAIS: IMPERATIVO, FUNCIONAL, RELACIONAL, POO

- **DADOS E OPERAÇÕES SÃO ENTIDADES DISTINTAS E DESLIGADAS, DECLARADAS POR ISSO EM ÁREAS DISTINTAS;**

(relembrar como se faz em ASSEMBLY, PASCAL, C, HASKELL, SQL, etc.)

- **PROGRAMAR => APLICAR OPERAÇÕES A DADOS TRANSFORMANDO-OS OU GERANDO RESULTADOS.**

este é o modelo  **$f(x)$**  »» operadores aplicados a operandos

**Exºs:**

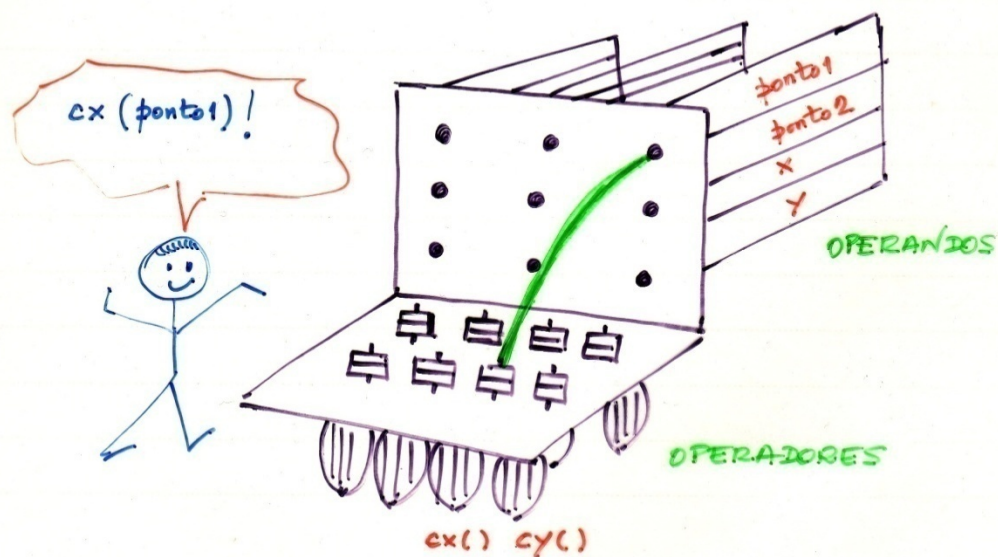
```
add x, y;  
println( sqrt(lado) );  
delete fich1
```

Em **POO** teremos que passar a pensar que **dados e operações** se definem de forma ligada; os dados possuem as suas próprias operações.

modelo  **$x.f()$**

- ESTE MODELO, ORIGINÁRIO DOS PRIMÓRDIOS DA COMPUTAÇÃO, EM QUE COMPUTADORES ERAM VISTOS COMO SUPER-CALCULADORAS, REALIZANDO POIS OPERAÇÕES SOBRE OPERANDOS, É VISÍVEL AINDA AOS MAIS DIVERSOS NÍVEIS:

NÍVEL MÁQUINA: INSTRUÇÕES + DADOS  
NÍVEL LINGUAGEM: EXPRESSÕES + VARIÁVEIS  
NÍVEL PROGRAMA: SUBROTINAS + ARGUMENTOS  
NÍVEL LING. COM.: COMANDOS + FICHEIROS



- Dados e operações são entidades separadas;
- Dados são entidades passivas sem operações directamente associadas;
- Programamos as ligações, ou seja, os  $f(x)$ ;

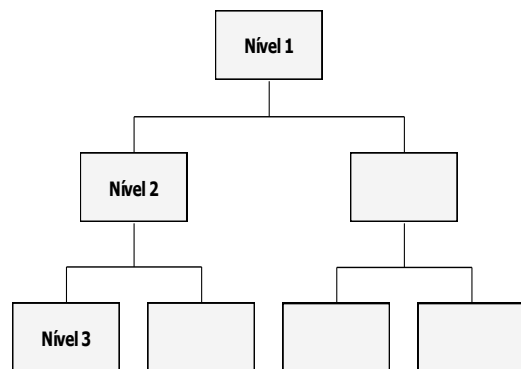
**Questão1: Como dividir os programas em módulos reutilizáveis ?**

▶ para não estar sempre a reinventar a roda e para << \$\$

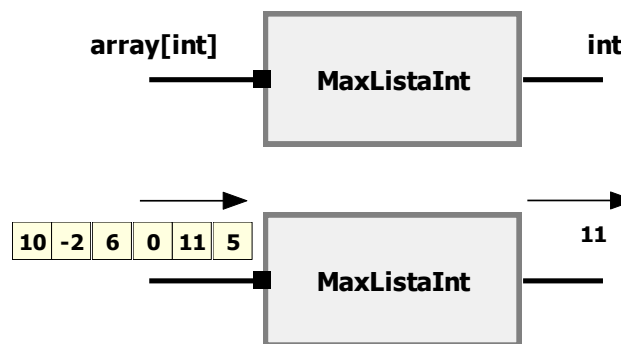
**Questão 2: Como controlar erros e modificações ?**

▶ os programas nunca estão prontos; estão sempre prontos para serem corrigidos e modificados; fáceis modificações implicam << \$\$

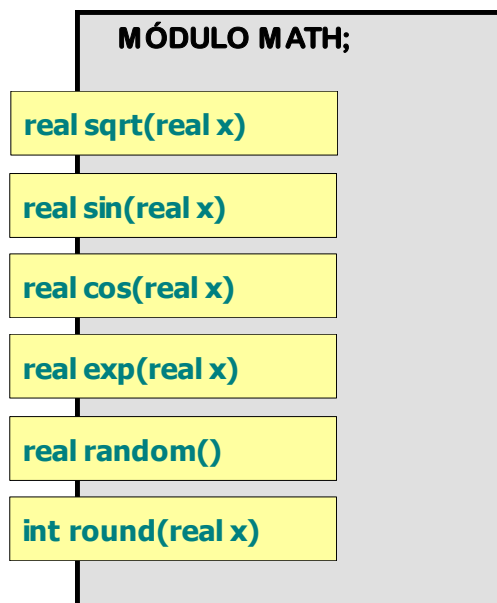
**Soluções tradicionais:**



**Refinamento Top-Down**



**Abstracção de Instruções (Procedimental)**



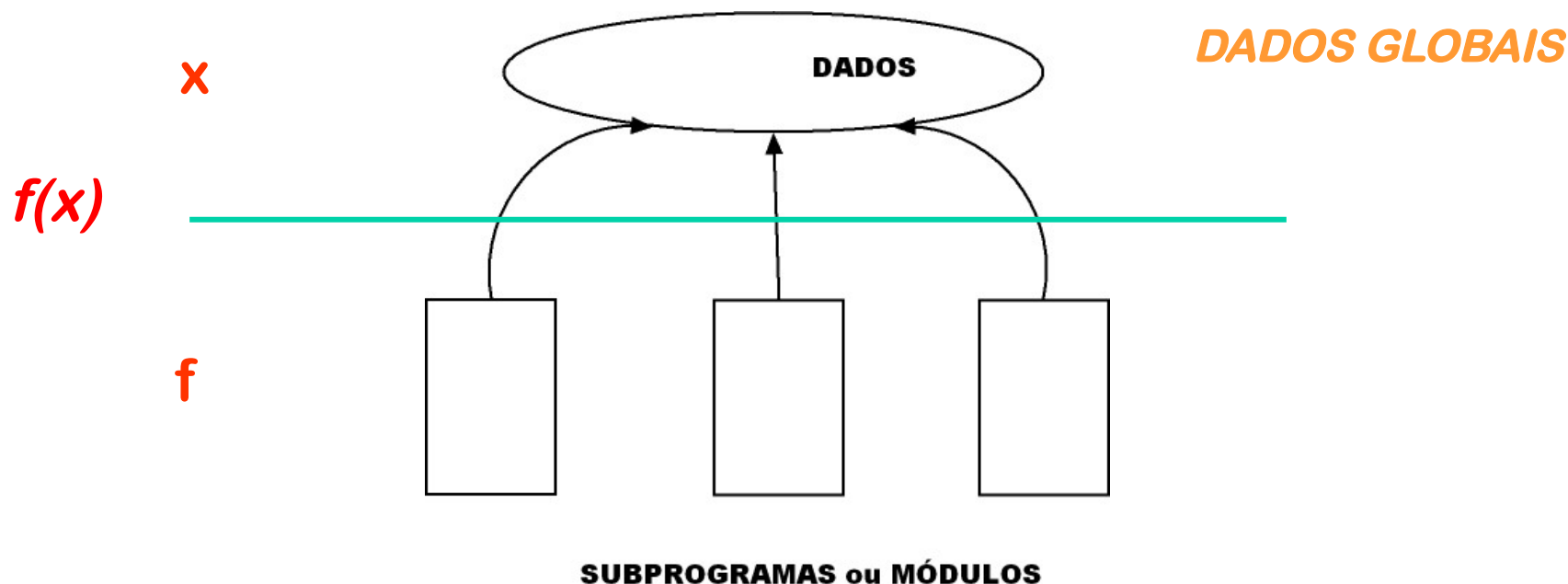
Módulos como **abstracções de instruções**, tal como em device drivers, módulo de cálculos matemáticos, de I/O, etc.

Assim, originalmente, a noção de **MÓDULO DE SOFTWARE** era a de que :



**MÓDULOS = ABSTRACÇÃO DE INSTRUÇÕES ou CONTROLO**

**PERMITEM:** ESTRUTURAÇÃO DE CÓDIGO, REUTILIZAÇÃO DE CÓDIGO, ABSTRACÇÃO, etc., MAS É PRECISO MAIS ...



► Exemplo estrutural de codificação imperativa típica e exemplo de má modularidade real porque **os dados são GLOBAIS !**

► Princípio de Sherlock Holmes: **Erro nos DADOS =>**  
Qual a instrução suspeita ? Neste exemplo **TODAS !**



## Calculadora (usa uma stack)

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

*definições e  
declarações  
comuns*

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
    sp++;
}
```

stack.c:

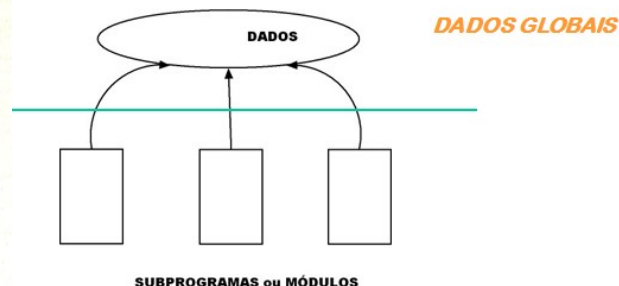
```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

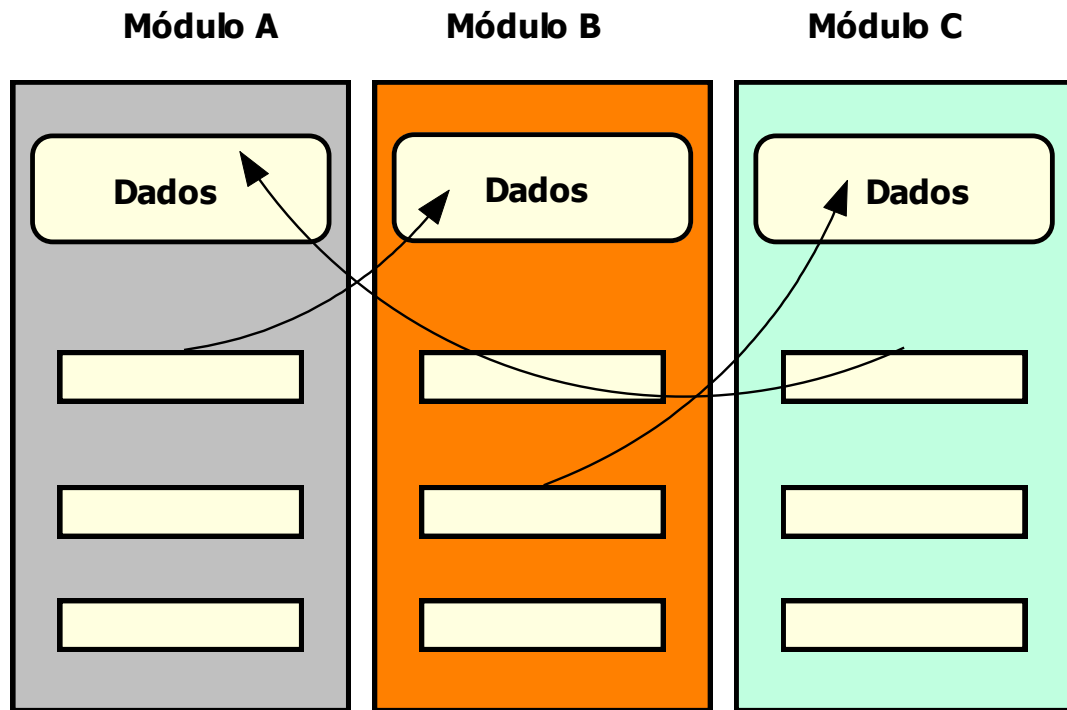
getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

- Programa está estruturado;
- Programa funciona;
- Mas os dados são **globais** !!

Não deveria  
ser possível !!





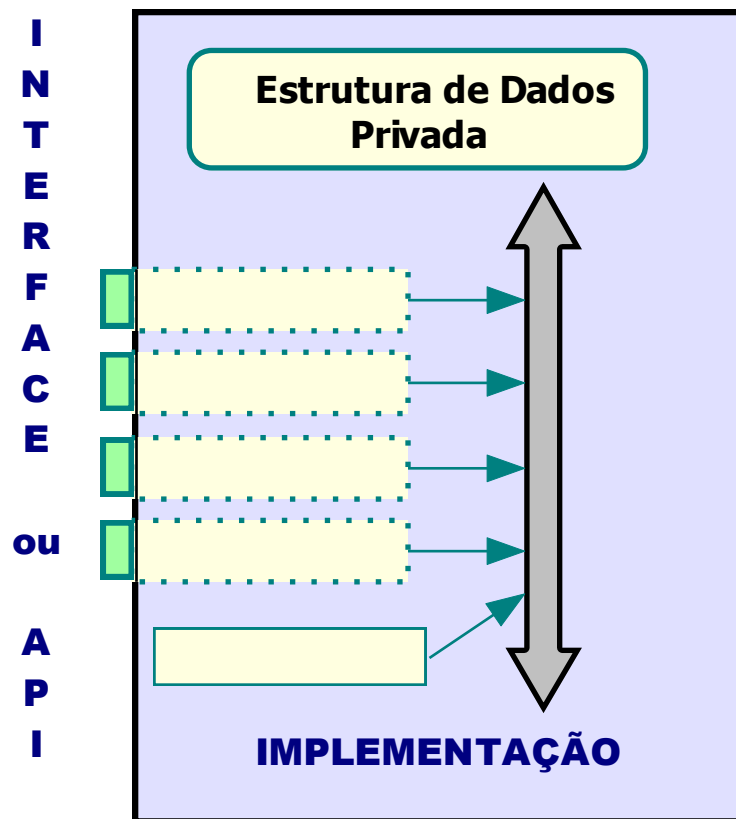
Se apenas pretendemos usar o módulo A, como A depende de B e B depende de C, teremos que os usar a TODOS.

- ▶ Estes módulos não são independentes;
- ▶ Dados de uns são acedidos por módulos externos;

**Solução: Módulo => Estrutura de Dados privada e suas operações**

**Módulo = Abstracção de Dados**

**Módulo = Interface + Implementação de Estrutura de Dados**



**MÓDULO É UMA CÁPSULA QUE CONTÉM UMA ESTRUTURA DE DADOS PRIVADA, NÃO ACESSÍVEL DO EXTERIOR, E AS ÚNICAS OPERAÇÕES QUE PODEM ACEDER A TAIS DADOS.**

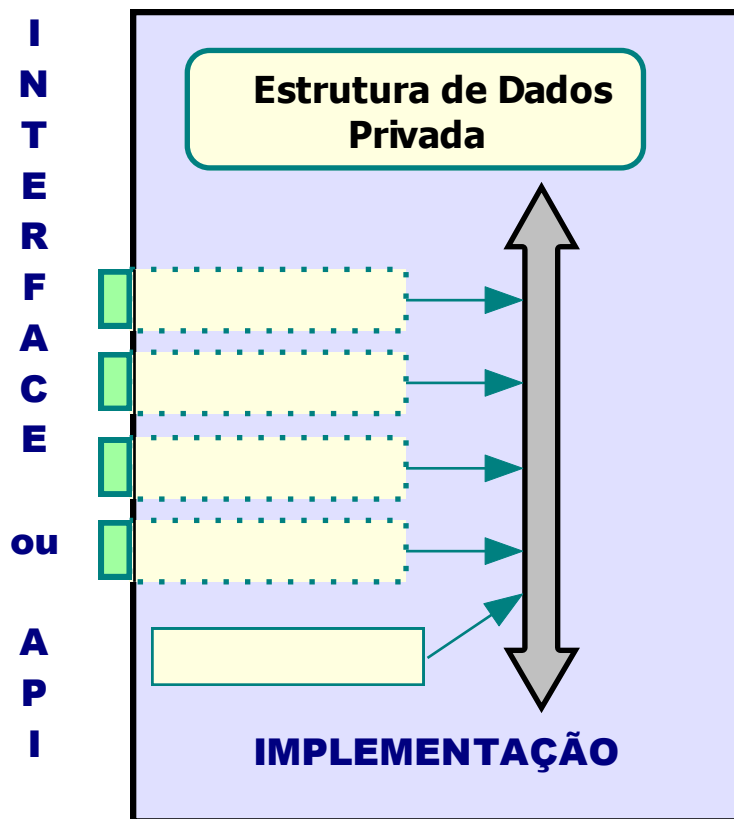
## ENCAPSULAMENTO DE DADOS

- Operações podem ser tornadas públicas, ou seja acessíveis do exterior, ou serem apenas internas ao módulo (**privadas**);
- Operações públicas formam a interface do módulo ie. o que pode ser invocado;



**Módulo = Abstracção de Dados**

**Módulo = Interface + Implementação de Estrutura de Dados**



- **API: Application Programmer's Interface**  
Operações que são acessíveis do exterior, ou seja, são tornadas **PÚBLICAS**;

- **ERROS:** Apenas o código interior ao módulo pode provocar erros nos dados (Sherlock Holmes tem agora a vida muito facilitada);

- **ABSTRACÇÃO:** a utilização do módulo não obriga (antes pelo contrário) ter que saber qual a representação interna, mas apenas a API; Black-Box de software;

- **REUTILIZAÇÃO:** módulo é independente e autónomo;

# SOLUÇÃO: ENCAPSULAMENTO

## Calculadora de stack

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

*definições e declarações comuns*

O encapsulamento pode ser garantido se as variáveis forem declaradas **static**

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
    sp++;
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

Agora, a instrução geraria um erro !

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Assim, podemos ter módulos de software reutilizáveis e protegidos, mesmo em C



Assim, em C, o encapsulamento básico pode ser garantido se as variáveis forem declaradas **static** tal como sugerido e aconselhado em manuais de C.

- As formas mais elaboradas de criação de módulos de dados em C deverão ser lidas, analisadas e posteriormente usadas, lendo os documentos seguintes colocados no BB de LI3.

### MODULARIDADE EM PROGRAMAS C

F. Mário Martins

LABORATÓRIOS DE INFORMÁTICA III - LEI – 2013/2014

### IMPLEMENTAÇÃO EM C DE ABSTRACÇÕES DE DADOS

#### TÉCNICA DOS TIPOS INCOMPLETOS

F. Mário Martins, LI3, 2015

## DESENVOLVIMENTO DE SOFTWARE EM LARGA ESCALA

### CONCEITOS FUNDAMENTAIS



**“DATA HIDING”**



**“IMPLEMENTATION HIDING”**



**ABSTRACÇÃO DE DADOS**



**ENCAPSULAMENTO**



**INDEPENDÊNCIA CONTEXTUAL**

**Compiladores não garantem verificação destas propriedades !!**



Dados privados e protegidos;

Representação dos dados não deve ser acedida directamente;

Acesso aos dados apenas usando a API;

As operações internas do módulo não devem possuir operações de I/O;



# LABORATÓRIOS DE INFORMÁTICA III

2015/2016

MIEI

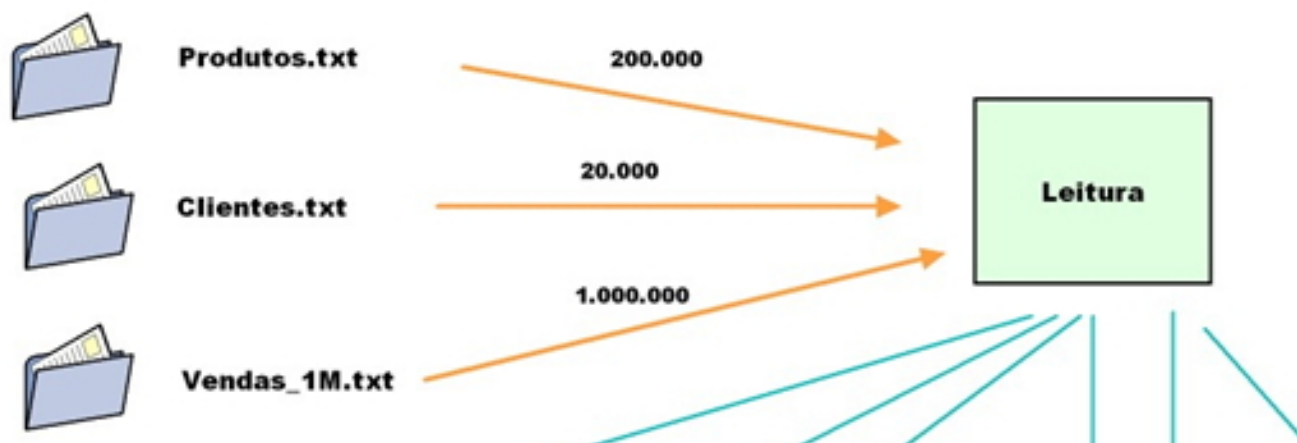
## TRABALHO PRÁTICO DE C

- ✓ Enunciado do problema;
- ✓ Requisitos de modularidade e funcionalidade;
- ✓ Estrutura do Relatório final;
- ✓ Avaliação: Critérios gerais.



## GereVendas: Gestão de Vendas de Hipermercado com 3 Filiais

☑ Pretende-se desenvolver uma aplicação em GNU C, com **código standard**, modular e eficiente, quer em termos de algoritmos quer em termos de estruturas de dados implementadas, que seja, antes de mais, capaz de ler e processar as linhas de texto de 3 ficheiros **.txt** indicados, um contendo todos os códigos de **produtos**, outro todos os códigos de **clientes** e o terceiro com registo de todas as **vendas feitas**.



Teremos 200.000 códigos de produtos, 20.000 códigos de clientes e 1.000.000 de compras.

No ficheiro **Produtos.txt** cada linha representa o **código de um produto** vendável no hipermercado, sendo cada código formado por duas letras maiúsculas e 4 dígitos (que representam um inteiro entre 1000 e 1999), cf. os exemplos,

AB1012    XY1185    BC1190

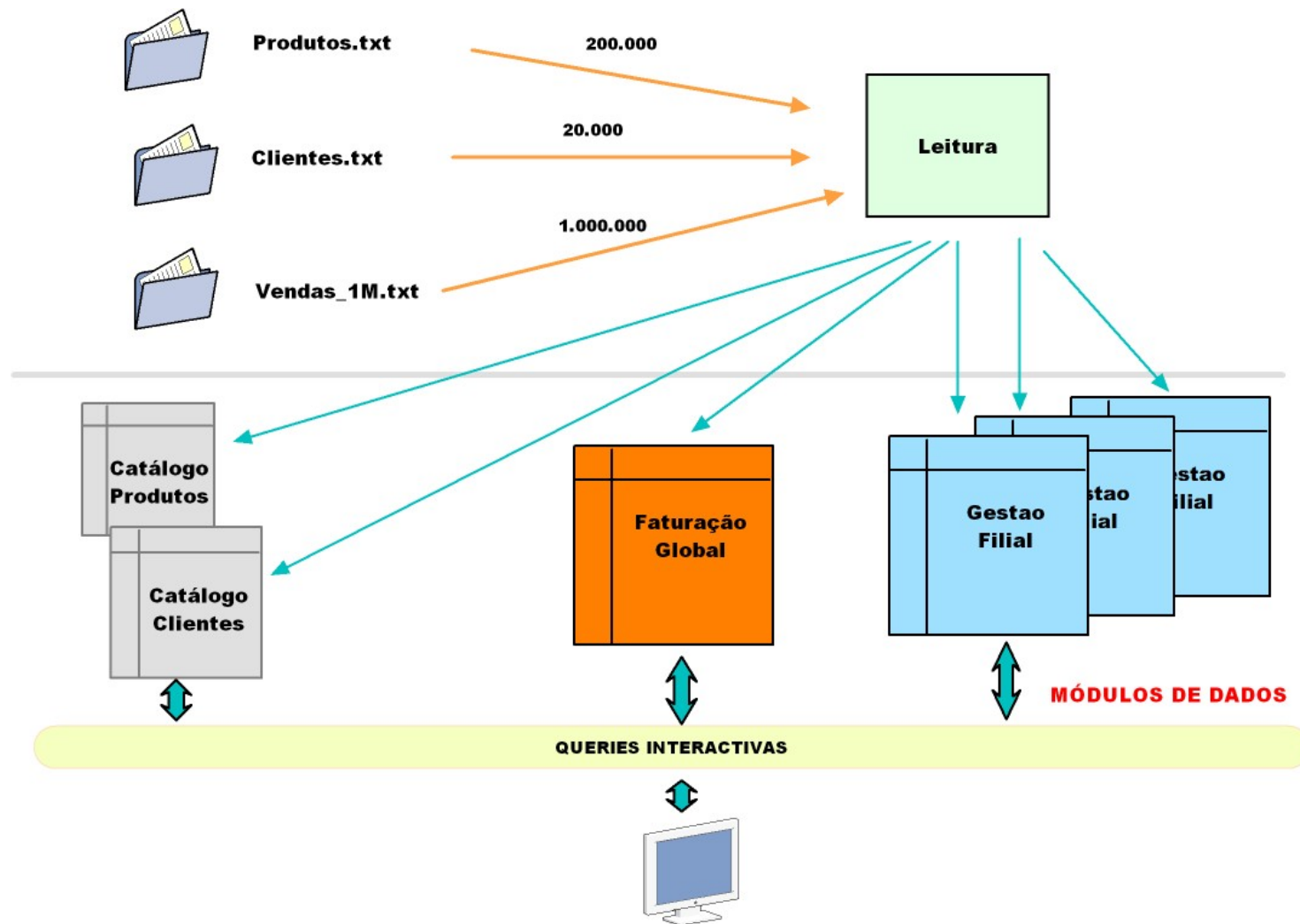
No ficheiro **Clientes.txt** cada linha representa o **código de um cliente** identificado no hipermercado, sendo cada código de cliente formado por uma letra maiúscula e 4 dígitos que representam um inteiro entre 1000 e 5000, cf.:

F2916    W1219    F2915

O ficheiro que será a maior fonte de dados do projecto designa-se por **Vendas\_1M.txt**, no qual **cada linha** representa o **registo de uma compra** efectuada no hipermercado. cf. os exemplos seguintes produto + preço unidade + nº de unidades + **Promoção** ou **Normal**, código cliente + mês + filial:

KR1583 77.72 128 P L4891 2 1    ← **registo de compra/venda**  
QQ1041 536.53 194 P X4054 12 3  
OP1244 481.43 67 P Q3869 9 1  
JP1982 343.2 168 N T1805 10 2

## Arquitectura da aplicação

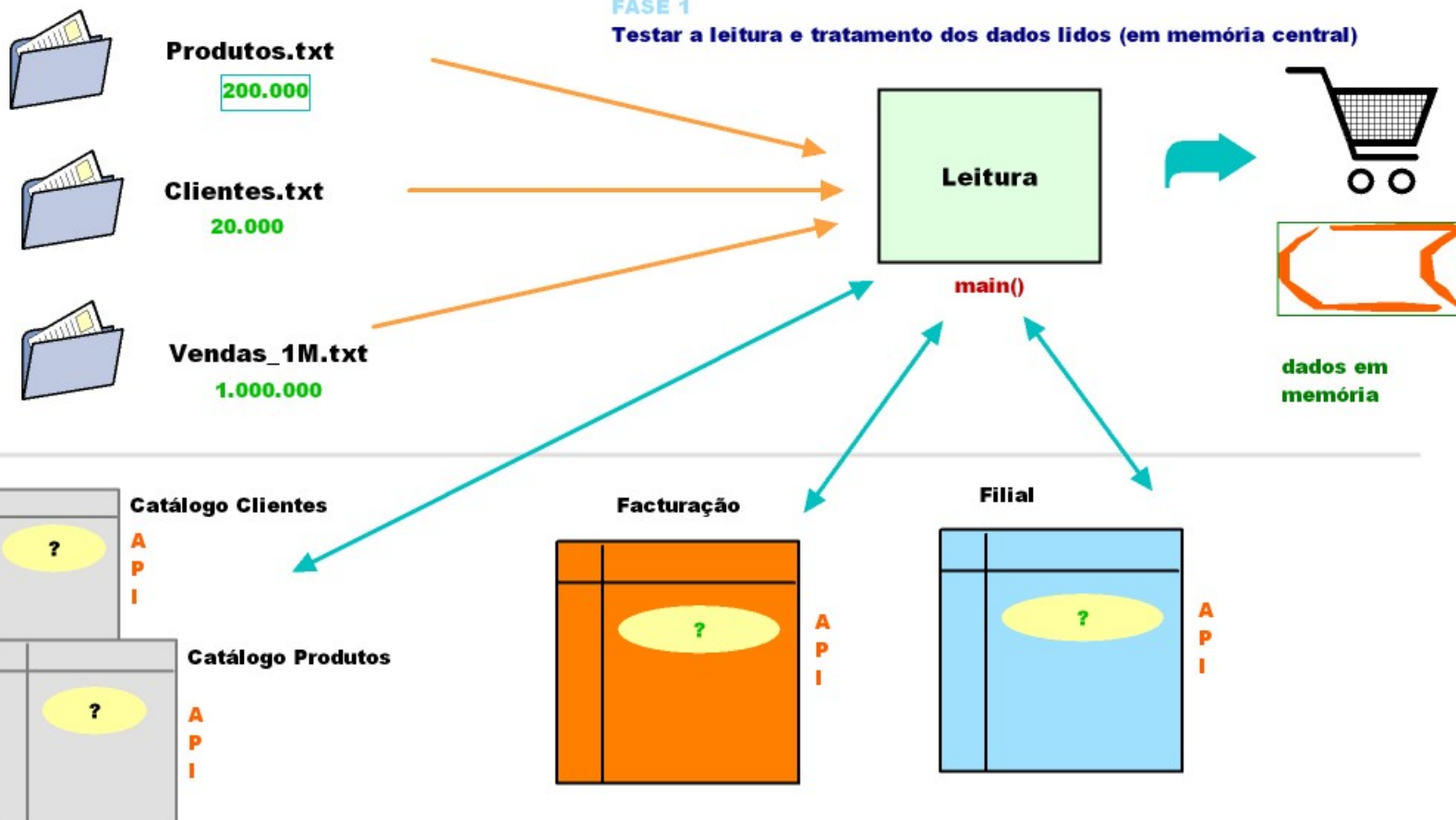




## FASE 1: Conselho

### FASE 1

Testar a leitura e tratamento dos dados lidos (em memória central)



### FASE 2

Começar a desenhar os módulos;  
Funções, públicas e privadas e  
Estruturas de dados, tendo em atenção as consultas;



## FASE 1: Ler e testar valores

\_\_\_\_\_ FIM DA LEITURA DE VENDAS\_1M \_\_\_\_\_

Produtos envolvidos: 171008

Clientes envolvidos: 16384

cf. Boletim Semanal/Quinzenal

Vendas efectivas: 1000000

Ultimo Cliente: A3439

Numero de Vendas Registadas para este cliente: ??

Numero de Vendas na Filial 1: ???

Numero de Vendas na Filial 2: ???

Linha mais longa: ???

Numero de Clientes com codigo começado por A: ???

Facturação Total registada: ???

### Queries interactivas.

Tendo sido apresentada a arquitectura genérica da aplicação, a efectiva estruturação de cada um dos módulos depende, naturalmente, da funcionalidade esperada de cada um deles. Tal é, naturalmente, completamente dependente das *queries* que a aplicação deve implementar para o utilizador final.

Deste modo, e fornecida que foi uma arquitectura de referência, deixa-se ao critério dos grupos de trabalho a concepção das soluções, módulo a módulo, para a satisfação da implementação de cada uma das *queries* que podem ser realizadas pelo utilizador e, até, a sua adequada estruturação sob a forma de menus, etc.

### Testes de performance.

Depois de desenvolver e codificar todo o projecto tendo por base o ficheiro **Vendas\_1M.txt**, dever-se-á realizar testes de *performance* e apresentar os respectivos resultados. Pretende-se comparar tempos de execução de certos *queries* usando os ficheiros **Vendas\_3M.txt (3 milhões de registos)** e **Vendas\_5M.txt (5 milhões de registos)**. Todos os ficheiros serão fornecidos numa pasta disponibilizada via BB.



### Requisitos para a codificação final.

A codificação final deste projecto deverá ser realizada usando a linguagem C e o compilador **gcc**. O código fonte deverá compilar sem erros usando o *switch* **-ansi**. Podem também ser utilizados outros *switches* de optimização, etc (cf. **-pedantic -O2 -Wall**). Para a correcta criação das *makefiles* do projecto aconselha-se a consulta do utilitário **GNU Make** no endereço [www.gnu.org/software/make](http://www.gnu.org/software/make).

Qualquer utilização de bibliotecas de estruturas de dados em C deverá ser sujeita a prévia validação por parte da equipa docente. Não são aceitáveis bibliotecas genéricas tais como LINQ e outras semelhantes.

O código final de todos os grupos será sujeito a uma análise usando a ferramenta **JPlag**, que detecta similaridades no código de vários projectos, e, quando a percentagem de similaridade ultrapassar determinados níveis, os grupos serão chamados a uma clara justificação para tal facto.



### Apresentação do projecto e Relatório.

O projecto será submetido por via electrónica num *site* do DI a indicar oportunamente (bem como o formato da pasta e a data e hora limite de submissão). Tal *site* garantirá quer o registo exacto da submissão quer a prova da mesma a quem o submeteu (via e-mail). Tal garantirá extrema segurança para todos.

O código submetido na data de submissão será o código efectivamente avaliado. A *makefile* deverá gerar o código executável, e este deverá executar correctamente.

**Projectos com erros de *makefile*, de compilação ou de execução serão de imediato rejeitados.**

---