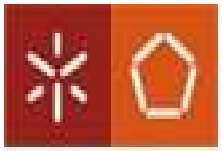


Universidade do Minho
Departamento de Informática

LABORATÓRIOS DE INFORMÁTICA III

2012/2013
LEI

2º ANO - 2º SEM

F. Mário Martins (fmm@di.uminho.pt)
João Miguel Fernandes (jmf@di.uminho.pt)
João Luís Sobral (jls@di.uminho.pt)

DI/UM

OBJECTIVOS

▣ **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, eventos e streams;

FUNCIONAMENTO

▣ **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. **Turnos = PL5 e PL2 (3ª. Feira) e PL4 (5ª.feira)**

Os alunos realizarão dois projectos práticos obrigatórios.

- O 1º projecto de C será de dimensão média e realizado de forma individual; terá 2 submissões calendarizadas.

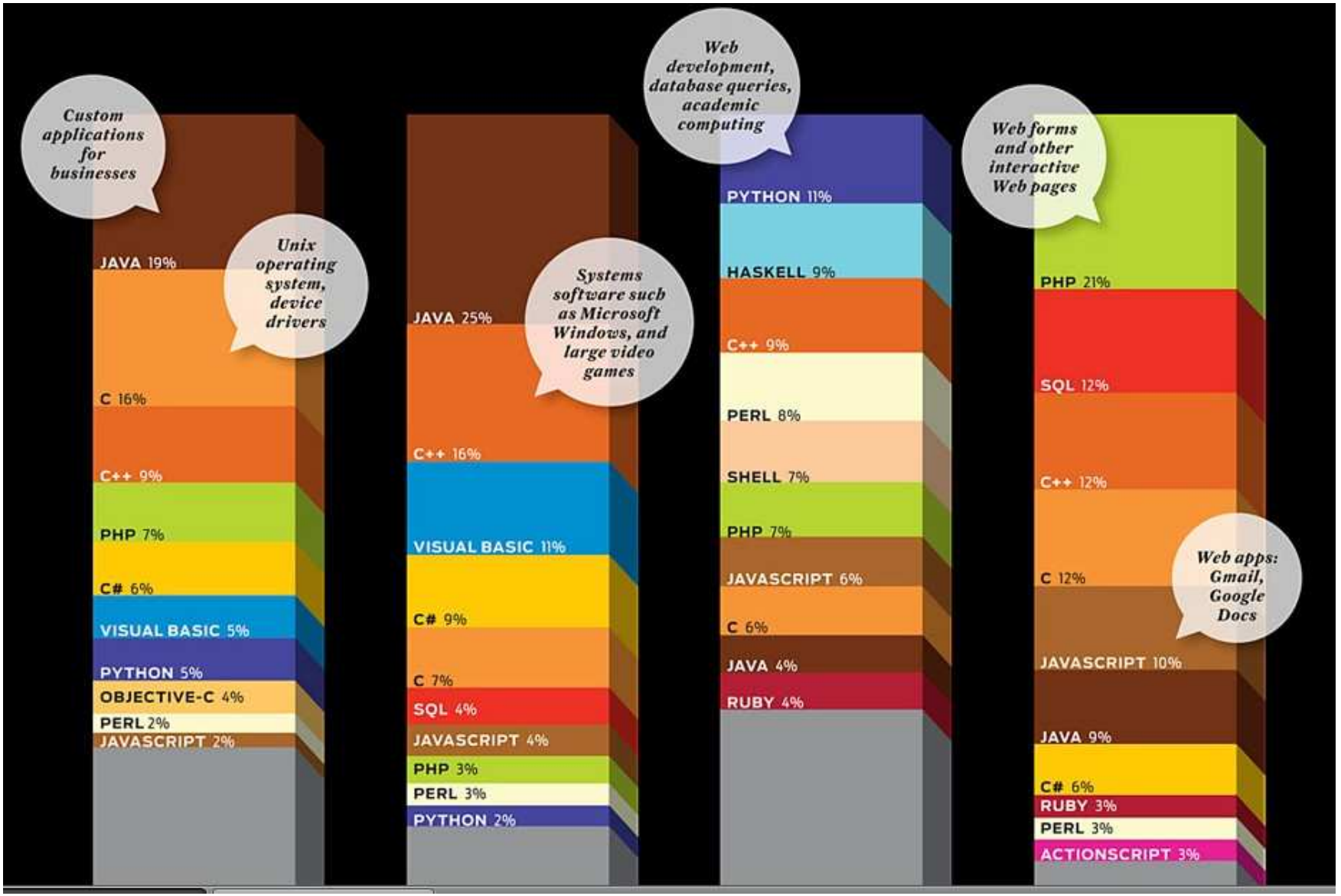
- O 2º projecto, de JAVA, será realizado em grupo (máx. 3 alunos) e terá apenas a submissão final.

- 1º projecto - **Linguagem C**
- 2º projecto - **Linguagem Java:** (em ambiente Robocode);

A fórmula que calcula a nota final e pressupõe :

Nota Final =60%*Proj1 + 40%*Proj2

PORQUÊ C e JAVA ?



CALENDÁRIO

CALENDÁRIO LI3
2012-2013

	Semana	2a.feira	3a.feira	4a. Feira	5a.feira	6a.feira	sábado	domingo	
1	18/02 a 24/02			COMUM					Aula comum de apresentação de LI3
2	25/02 a 02/03								
3	04/03 a 09/03								
4	11/03 a 16/03								Entrega electrónica Fase1 de C
5	18/03 a 23/03								
	25/03 a 30/03								
6	01/04 a 06/04								
7	08/04 a 13/04								
8	15/04 a 20/04								
9	22/04 a 27/04			COMUM					Entrega electr. Fase2 de C / Aula apresent. Java
10	29/04 a 04/05								
11	06/05 a 11/05								Avaliações de C
	13/05 a 18/05								
12	20/05 a 25/05								
13	27/05 a 01/06								
14	03/06 a 08/06								Entrega electrónica dos trabalhos de Java
15	10/06 a 15/06								Avaliações de Java
	17/06 a 22/06								
	24/06 a 29/06								
	01/07 a 06/07								
	08/07 a 13/07								

- Aula comum em sala a marcar
- entregas electrónicas de trabalhos
- avaliações dos trabalhos
- laboratórios de C
- laboratórios de Java
- férias feriados
- queima
- recurso

TURNOS EFECTIVOS

ACESSO BB : LI3CJAVA

TURNOS DISPONÍVEIS (inscrições são desnecessárias):

3ª. Feira, 14H00-16H00 (PL5)

3ª. Feira, 16H00-18H00 (PL2)

5ª. Feira, 14H00-16H00 (PL4)

PARADIGMAS

■ 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.

PARADIGMA IMPERATIVO

PARADIGMAS TRADICIONAIS: IMPERATIVO, FUNCIONAL, RELACIONAL

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

(relembrar como se faz em ASSEMBLY, PASCAL, C, HASKELL, BDs, 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
```

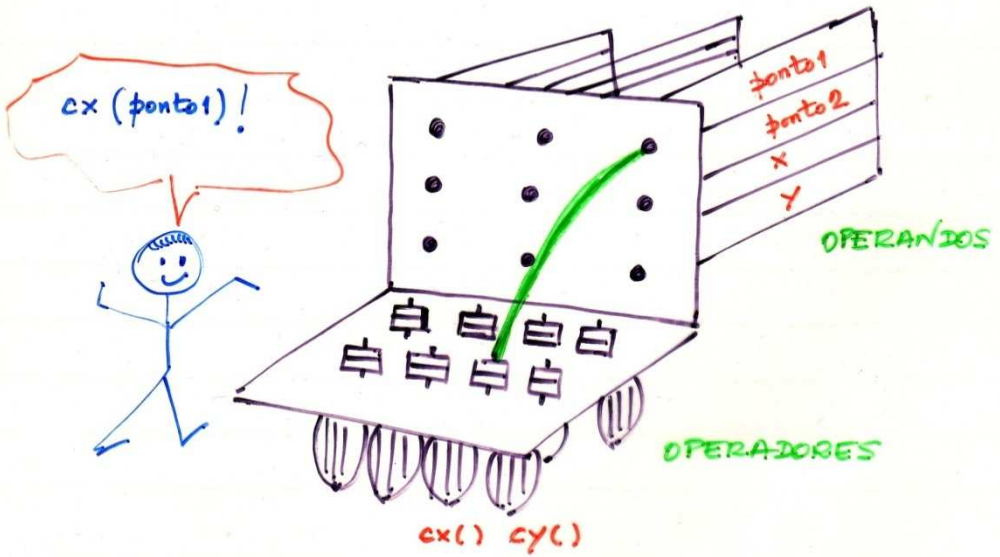
Em POO teremos que passar a pensar que dados e operações se definem de forma ligada; os dados possuem as suas próprias operações.

modelo x.f()

PARADIGMA IMPERATIVO

- 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);**

MODULARIDADE: Problemas

Questão1: Como dividir os programas em módulos reutilizáveis ?

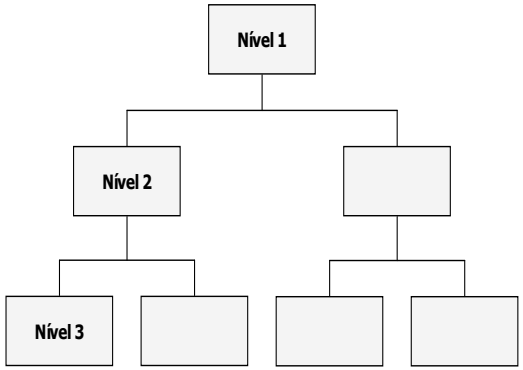
- para não estar sempre a reinventar a roda e para << $$

Questão 2: Como controlar erros e modificações ?

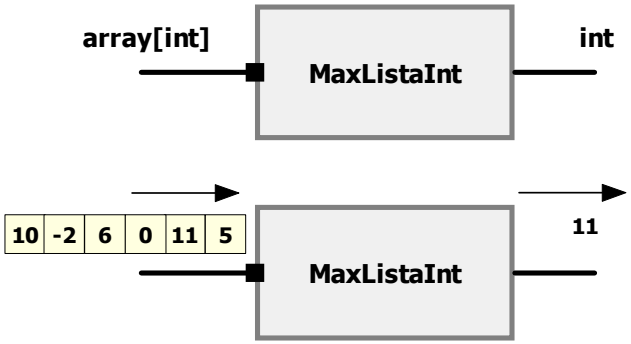
- os programas nunca estão prontos; estão sempre prontos

para serem corrigidos e modificados; fáceis modificações implicam << $$

Soluções tradicionais:

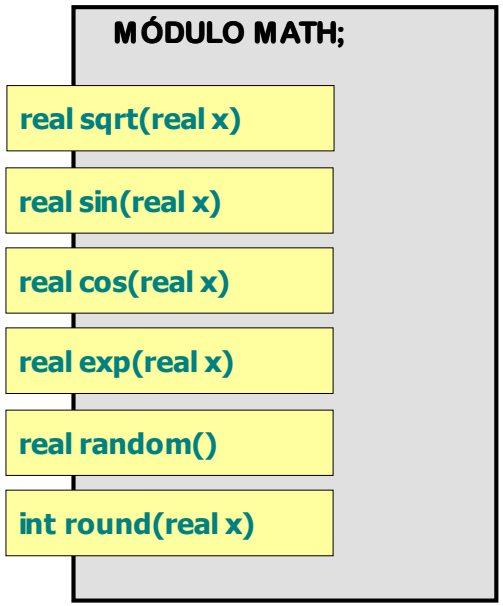


Refinamento Top-Down



Abstracção de Instruções (Procedimental)

MODULARIDADE: Problemas



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

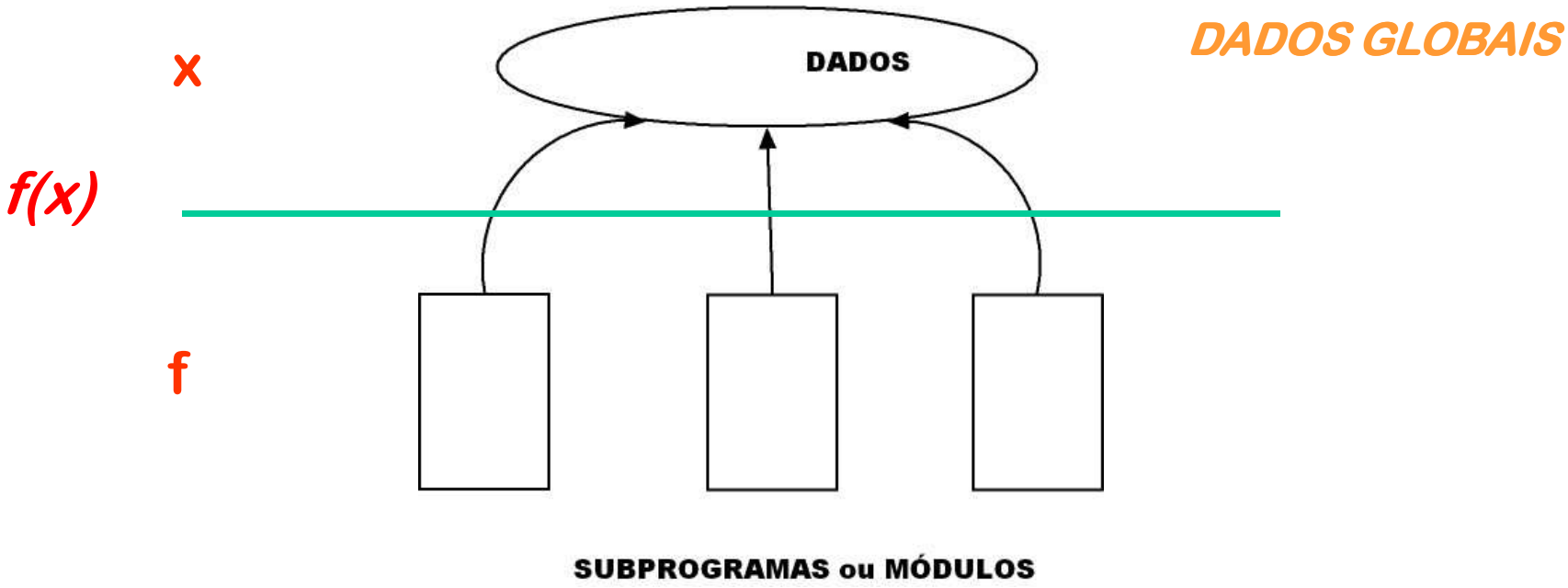
Assim, originalmente, a noção de **MÓDULO DE SOFTWARE** era a de que :

↓

MÓDULOS = ABSTRACÇÃO DE INSTRUÇÕES ou CONTROLO

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

MODULARIDADE: Esquema 1



▶ 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 !

MODULARIDADE: Exemplo em C

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:

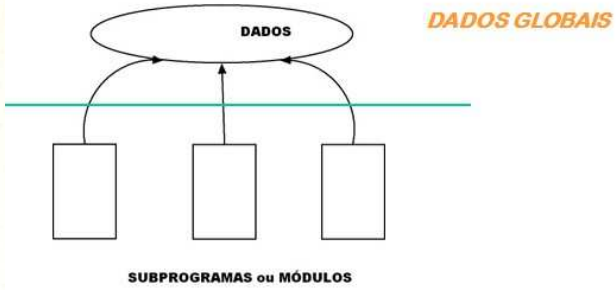
```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c:

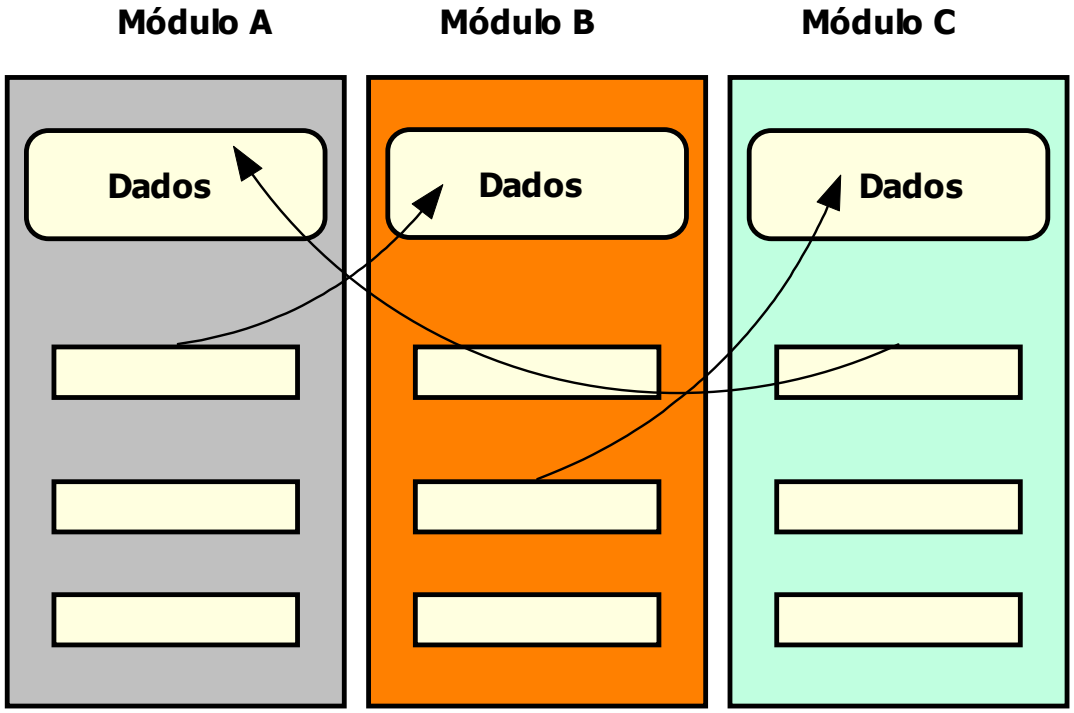
```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Não deveria ser possível !!

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



MODULARIDADE: Esquema 2



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

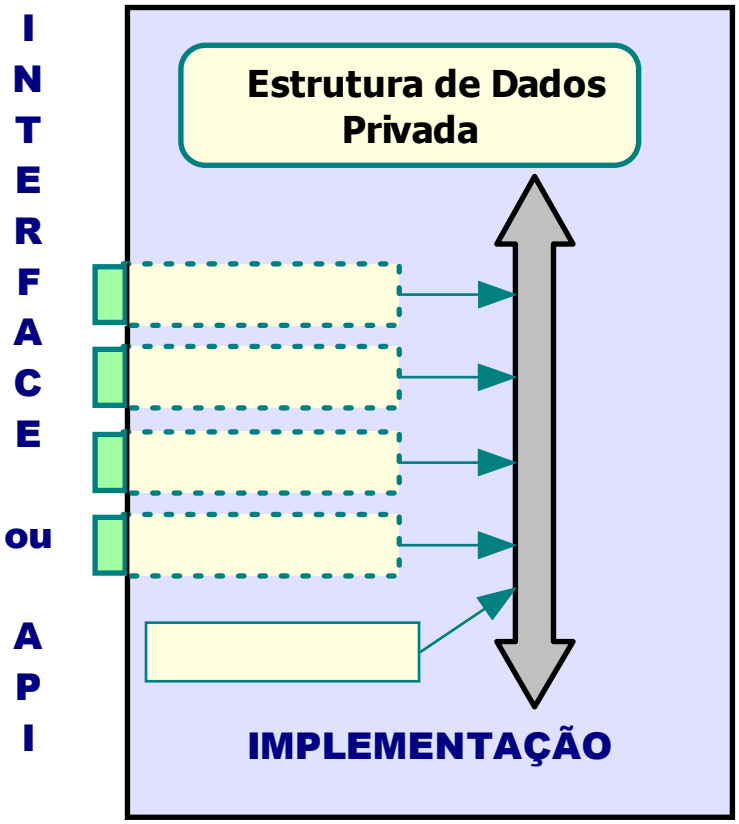
- Estes módulos não são independentes;
- Dados de uns são acedidos por módulos externos;

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

SOLUÇÃO: ENCAPSULAMENTO

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

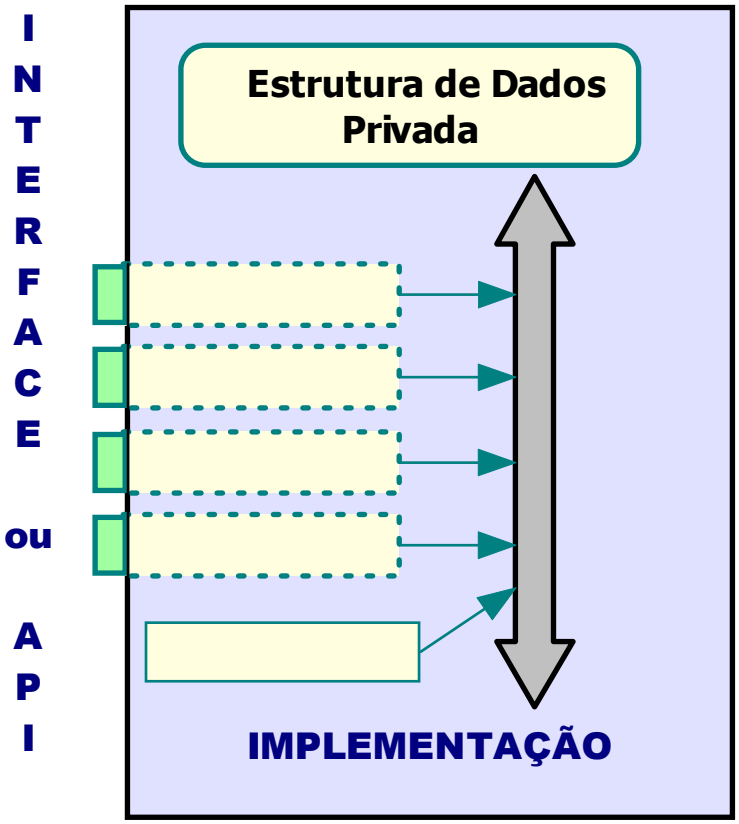
- **Operações podem ser tornadas públicas, ou seja acessíveis do exterior, ou serem apenas internas ao módulo (privadas);**

- **Operações públicas formam a interface do módulo ie. o que pode ser invocado;**

SOLUÇÃO: ENCAPSULAMENTO

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**;

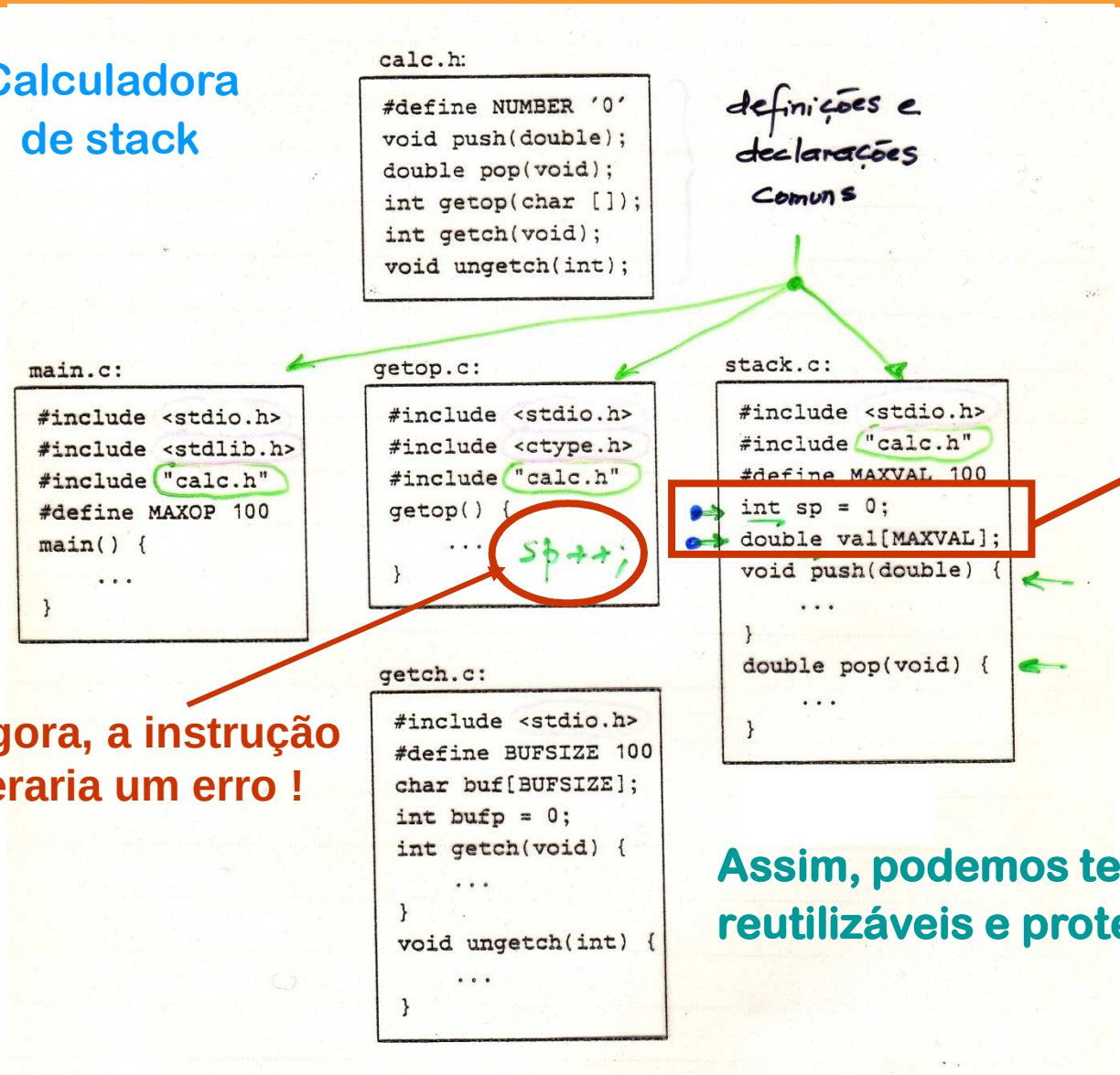
- ERROS: Apenas o código interior ao módulo pode provocar erros nos dados (Sherlock Holmes tem agora a vida muito facilitada);

- 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



O encapsulamento pode ser garantido se as variáveis forem declaradas static

Agora, a instrução geraria um erro !

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

ENCAPSULAMENTO EM C

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

Static storage class designation can also be applied to external variables. The only difference is that static external variables can be accessed as external variables only in the file in which they are defined. No other source file can access static external variables that are defined in another file.

```
/* File: xxx.c */
    static int count;
    static char name[8];
    main()
    {
        ... /* program body */
    }
```

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;