Universidade do Minho
Departamento de Informática

LABORATÓRIOS DE INFORMÁTICA III

2014/2015

LEI

2º ANO - 2º SEM

Aula Comum Nº 2

Questões finais sobre TP de C

Submissão do TP de C

F. Mário Martins (fmm@di.uminho.pt)
António Luís de Sousa (als@di.uminho.pt)

DI/UM

CALENDÁRIO

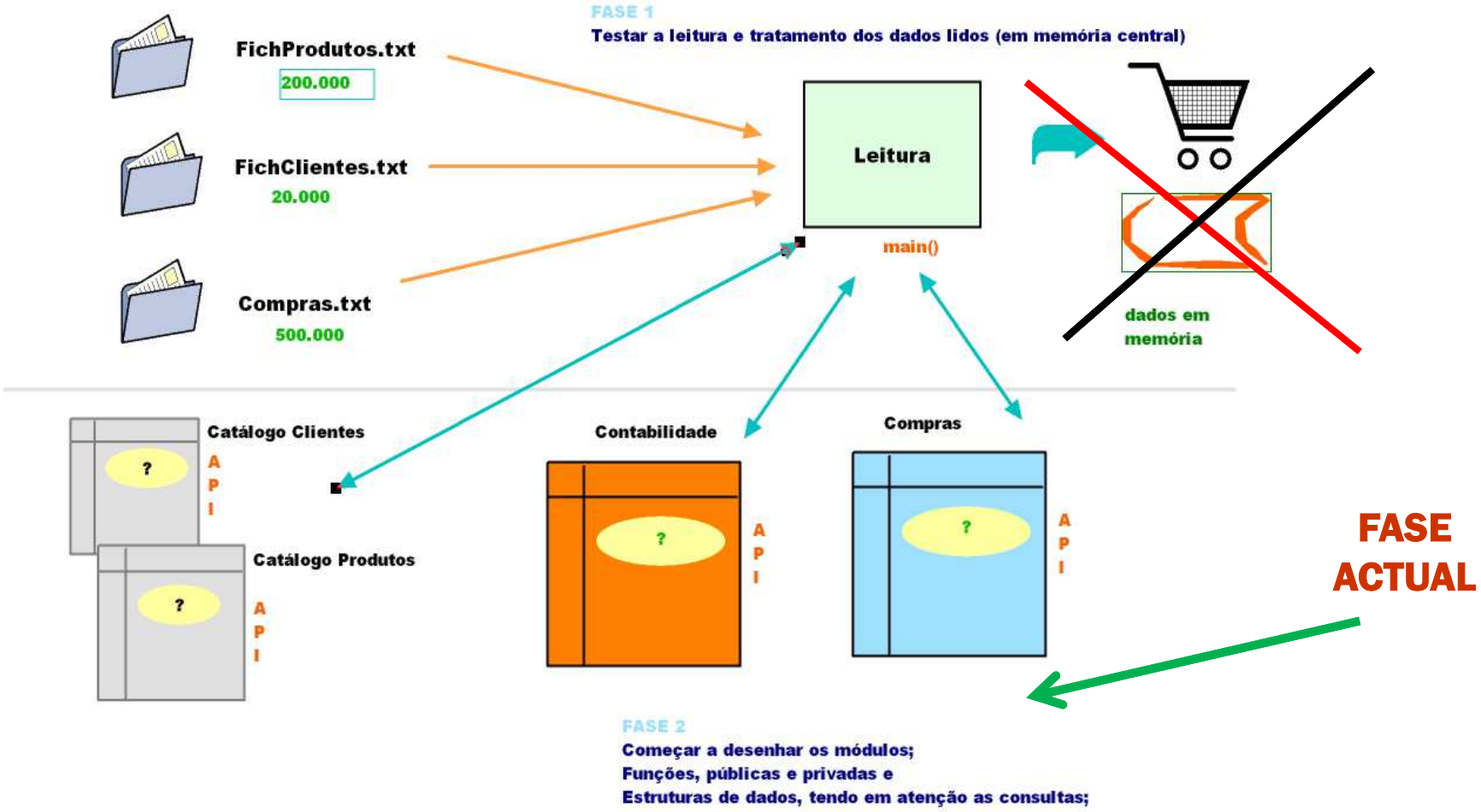
CALENDÁRIO LI3
2014-2015

Semana	2a.feira	3a.feira	4a. Feira	5a.feira	6a.feira	sábado	
16/02 a 21/02							Semana de LEI
23/02 a 28/02			COMUM				Aula comum de apresentação de LI3 (TP de C)
02/03 a 07/03							
09/03 a 14/03							
16/03 a 21/03							
23/03 a 28/03							
30/03 a 04/04							Páscoa
06/04 a 11/04			COMUM				
13/04 a 18/04							
20/04 a 25/04			COMUM				TP de Java + Entrega electrónica do TP de C
27/04 a 02/05							Avaliações do TP de C
04/05 a 09/05							
11/05 a 16/05							*Semana da queima*
18/05 a 23/05			COMUM				
25/05 a 30/05							
01/06 a 06/06							
08/06 a 13/06							Entrega electrónica do TP de Java
15/06 a 20/06							Avaliações do TP de Java
22/06 a 27/06							Lançamento das Notas Finais
29/06 a 04/07							
06/07 a 11/07							

Estamos AQUI !

- 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

Arquitectura da aplicação



Projecto de C: GESTHIPER

✍ Importância de realizar algum profiling quando se desenvolvem aplicações de grande escala, em especial quanto aos dados; Como fazer? Testar os dados e fazer algumas medidas; **FASE 1**

☑ Exemplos:

- nº de linhas em Compras.txt → 500000
- nº de compras válidas em Compras.txt → 445344
- buffer para fgets(s, ?, fich) → 32/64
- códigos de clientes errados → 52283
- códigos de produto errados → 2373
- total de compras de valor 0.0 → 247
- códigos de produtos por letra (média) → +- 17000
- códigos de cliente por letra aceitável (média) → +- 2700

Projecto de C: GESTHIPER

Importância de ter conhecimentos seguros sobre coisas básicas;

- Existe alguma confusão entre char [], char* e o conceito de "string" em C;
- Uma "string" é o prefixo de um char [] **"terminado" por** '\0' (delimitador obrigatório);
- Um char [] pode conter várias "strings" !
- char* strcpy(char [], char*) char* strcat(char* , char*) char* strdup(char*)

```
char *
strcpy(char *s1, const char *s2)
{
    char *s = s1;
    while ((*s++ = *s2++) != 0)
        ;
    return (s1);
}
```

(Não aloca espaço)

```
char *
strdup(const char *str)
{
 size_t siz;
 char *copy;

 siz = strlen(str) + 1;
 if ((copy = malloc(siz)) == NULL)
        return(NULL);
 (void)memcpy(copy, str, siz);
 return(copy);
}
```

(Aloca espaço e copia)

- Atenção: malloc() não inicializa; memset() não aloca; calloc() = malloc() + memset() ;
- Má compreensão destes mecanismos => segmentation faults;
- Não descurar nunca o uso de memcopy() e realloc(); Nunca usar bcopy();

Projecto de C: GESTHIPER

Importância de ter conhecimentos seguros sobre coisas básicas;

strncpy(3) - Linux man page

Name

strcpy, strncpy - copy a string

Synopsis

```
#include <string.h>

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

Description

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy**() writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

A simple implementation of **strncpy**() might be:

```
char *
strncpy(char *dest, const char *src, size_t n)
{
    size_t i;

   for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';

   return dest;
}
```

Return Value

The **strcpy**() and **strncpy**() functions return a pointer to the destination string *dest*.

Pelos vistos strncpy() está na moda.
Porquê ?

Importância de ter conhecimentos seguros sobre coisas básicas;

12

One major difference between `strtok()` and `strsep()` is that `strtok()` is standardized (by the C standard, and hence also by POSIX) but `strsep()` is not standardized (by C or POSIX; it is available in the GNU C Library, and originated on BSD). Thus, portable code is more likely to use `strtok()` than `strsep()`.

Stack vs Heap

So far we have seen how to declare basic type variables such as `int`, `double`, etc, and complex types such as arrays and structs. The way we have been declaring them so far, with a syntax that is like other languages such as MATLAB, Python, etc, puts these variables on the **stack** in C.

The Stack

What is the stack? It's a special region of your computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "FILO" (first in, last out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, **all** of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

The advantage of using the stack to store variables, is that memory is managed for you. You don't have to allocate memory by hand, or free it once you don't need it any more. What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

A key to understanding the stack is the notion that **when a function exits**, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables. A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

Another feature of the stack to keep in mind, is that there is a limit (varies with OS) on the size of variables that can be store on the stack. This is not the case for variables allocated on the **heap**.

To summarize the stack:

- the stack grows and shrinks as functions push and pop local variables
- there is no need to manage the memory yourself, variables are allocated and freed automatically
- the stack has size limits
- stack variables only exist while the function that created them, is running

The Heap

The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use **pointers** to access memory on the heap. We will talk about pointers shortly.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Projecto de C: GESTHIPER

Importância de ter conhecimentos seguros sobre coisas básicas;

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void) {
 char** strarray = NULL;
 int i = 0, strcount = 0;
 char line[1024];

 while( (fgets(line, 1024, stdin)) ) {
        strarray = (char **) realloc(strarray, (strcount + 1) * sizeof(char *));
        strarray[strcount++] = strdup(line);
 }

 /* Imprimir o array de strings */
for(i = 0; i < strcount; i++)
 printf("strarray[%d] == %s", i, strarray[i]);

 /*
 // Libertar o array de strings
 // Nota: Primeiro libertar o espaço de cada string !!
 */
 for(i = 0; i < strcount; i++) free(strarray[i]);
 free(strarray);
 return 0;
}
```

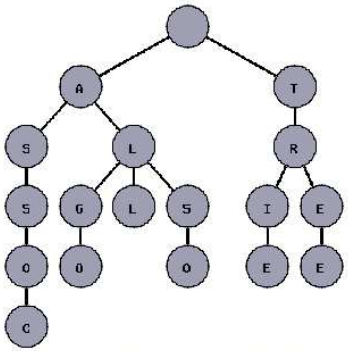
Arrays dinâmicos em C
(apenas um exemplo em que não se sabe à partida quantas strings vão ser lidas)

Análise das Consultas

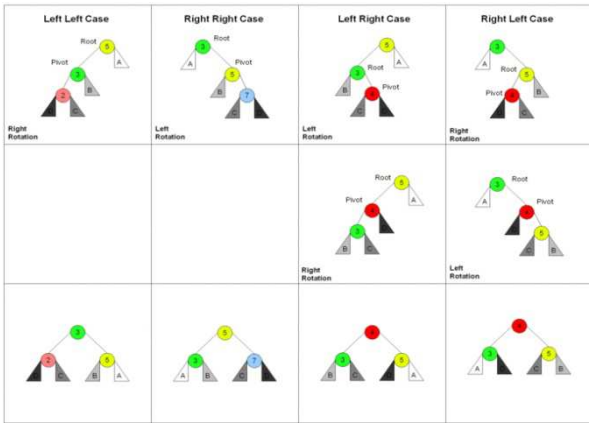
Q?	CProds	CClientes	Contab	Compras	Acesso	Crono
Q2	✓				por letra A .. Z	
Q3			✓		mês + cod. Produto	
Q4			✓			
Q5				✓	cod. Cliente	
Q6		✓			por letra A .. Z	
Q7			✓		intervalo de meses	
Q8				✓	cod. Produto	
Q9				✓	mês + cod. Cliente	
Q10				✓		
Q11			✓	✓		
Q12				✓		
Q13				✓	cod. Cliente	
Q14			✓	✓		

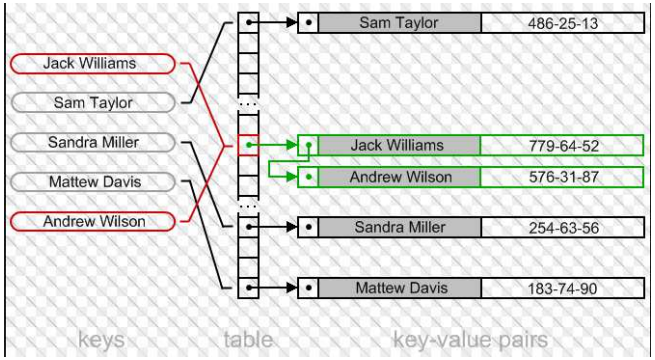
Estruturas de Implementação dos Módulos

The next figure shows a trie with the words "tree", "trie", "algo", "assoc", "all", and "also."



Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.





Bibliotecas Disponíveis – necessitam de ajustes

GNU libavl - Summary

Group Main Homepage Download Mailing lists Source code Bugs News

This project is part of the GNU Project.

GNU libavl is a library in ANSI/ISO C for the manipulation of binary trees and balanced binary trees. libavl is written using a literate programming system called TexiWEB. By way of TexiWEB, libavl is as much a textbook on binary trees and balanced binary trees as it is a collection of code.

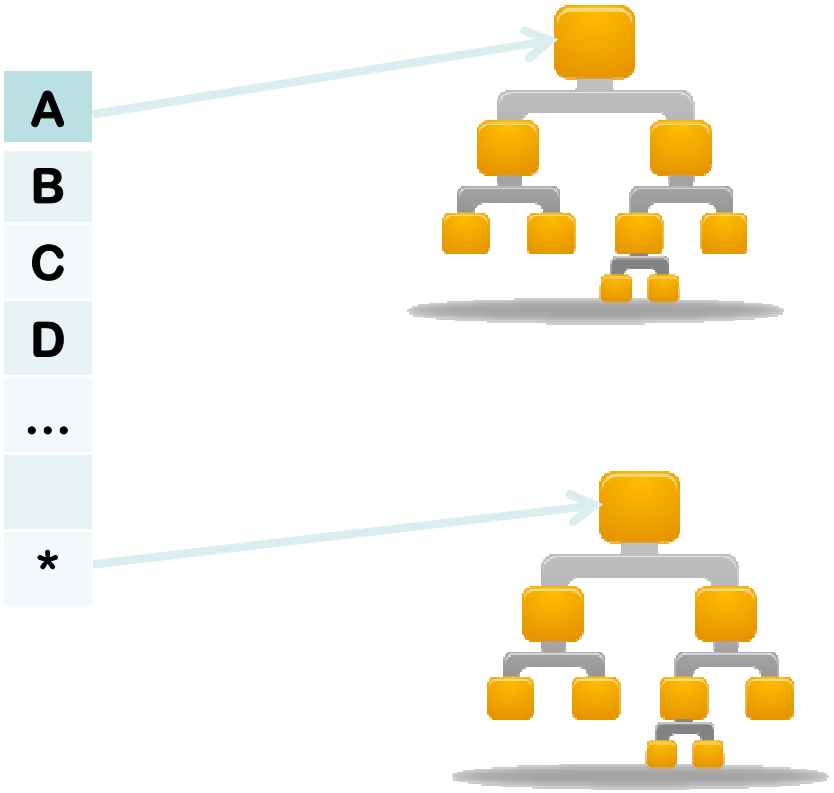
libavl supports the following kinds of trees:

- Plain binary trees:
  - Binary search trees
  - AVL trees
  - Red-black trees
- Threaded binary trees:
  - Threaded binary search trees
  - Threaded AVL trees
  - Threaded red-black trees
- Right-threaded binary trees:
  - Right-threaded binary search trees
  - Right-threaded AVL trees
  - Right-threaded red-black trees
- Binary trees with parent pointers:
  - Binary search trees with parent pointers
  - AVL trees with parent pointers
  - Red-black trees with parent pointers

Registration Date: Sat 24 Feb 2007 10:03:42 PM UTC
License: GNU Lesser General Public License
Development Status: 5 - Production/Stable

☑ FASE 2: Pensar na arquitectura de cada módulo em função das queries requisitadas pelo projecto. Pensar na estrutura de dados interna ao módulo e pensar nas funções que serão disponibilizadas aos clientes dos módulos (o cliente principal será o main()).

EXEMPLO:



☑ FASE 2: Declarações .h

BinarySearchTree.h

```
typedef int ElementType;

#ifndef _BINARY_SEARCH_TREE_H
#define _BINARY_SEARCH_TREE_H
struct TreeNode;
typedef struct TreeNode* Position;
typedef struct TreeNode* SearchTree;

SearchTree MakeEmpty(SearchTree T);
Position Find(ElementType X,SearchTree T);
Position FindMin(SearchTree T);
Position FindMax(SearchTree T);
SearchTree Insert(ElementType X,SearchTree T);
SearchTree Delete(ElementType X,SearchTree T);
ElementType Retrieve(Position P);
#endif
```

Tipos Opacos

Tipos incompletos de C

A qualidade e extensão de uma API (.h em C) é muito relevante para a utilidade do módulo desenvolvido.

Módulos: Tipos de Dados criados, API e utilização

CatalogoClientes.h

```
typedef char* CodCliente; (???)
typedef struct cat_clientes* CatalogoClientes;
/* API */
CatalogoClientes inicializa_CatalogoClientes();
CatalogoClientes insereCliente(CatalogoClientes catcli, CodCliente codcli);
BOOLEAN existeCliente(CatalogoClientes catcli, CodCliente codcli);
char** listaPorLetra(CatalogoClientes catcli, char letra);
CatalogoClientes removeCliente(CatalogoClientes catcli, CodCliente codcli);
……
```

Boolean.h

```
typedef int BOOLEAN;
#define TRUE  1
#define FALSE 0
```

Projecto de C: GESTHIPER

Compras.h

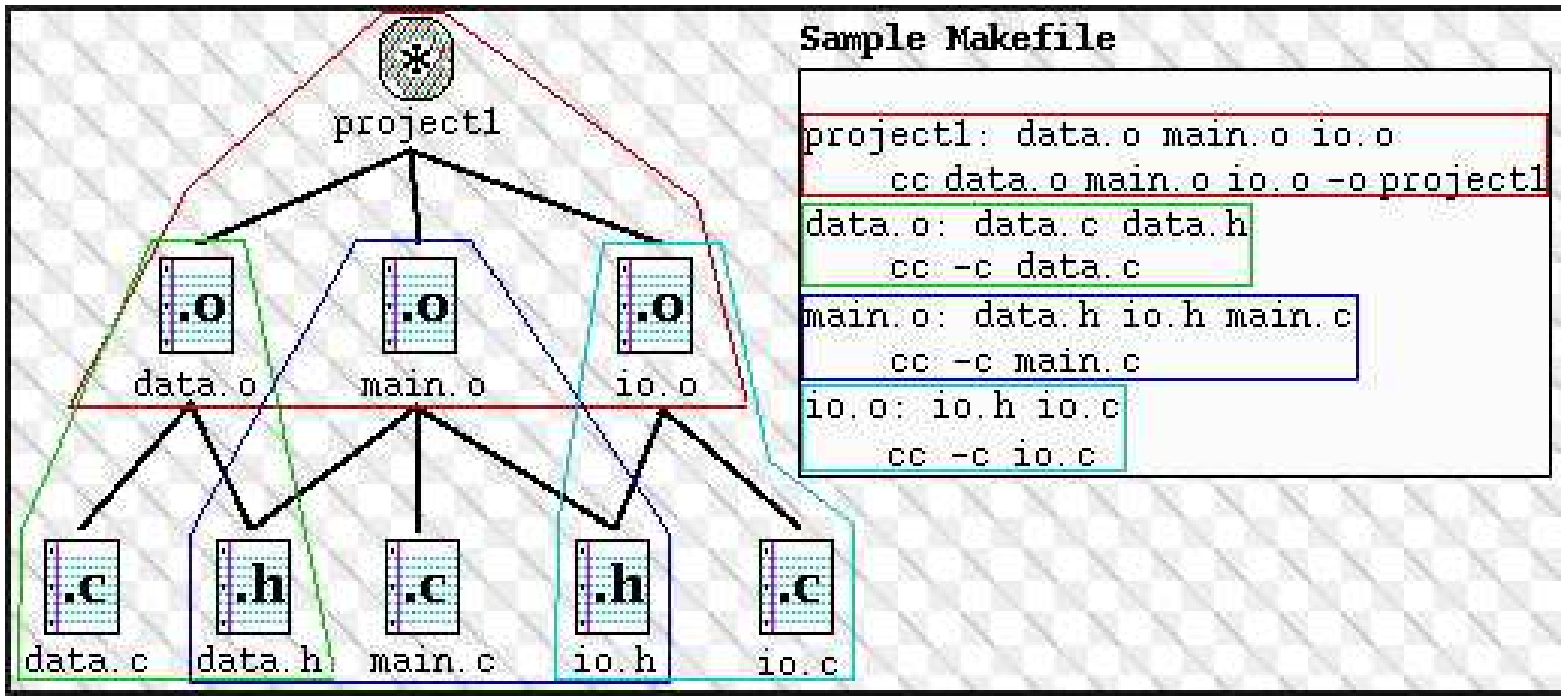
```
typedef struct compra* Compra;
typedef struct compras* Compras;
…….
/* API Compra */
Compra criaCompra(char* codProd, char* codCli, int unid, double preco, ….);
double getPreco(Compra cp);
……
/* API Compras */
Compras inicializa_Compras();
Compras insereCompra(Compras c, Compra cp);
…….
```

Projecto de C: GESTHIPER

main.c

```
CatalogoClientes catCli = inicializa_CatalogoClientes();
CatalogoProdutos catProds = inicializa_CatalogoProdutos();
Compras compras = inicializa_Compras();
Contabilidade contab = inicializa_Contabilidade();
……….
        catCli = insereCliente(catCli, cliente);
…….
```

Makefile e Grafo de Dependências



Projecto de C: Relatório a apresentar

☑ Capa do projecto com identificação do grupo BB, nomes, números e fotografia dos elementos do grupo;

☑ Para cada módulo, apresentar um desenho comentado da estrutura de dados que o implementa, todos os typedef respectivos (quer no .h quer no .c), a API comentada, função a função (o que faz cada uma), e, para cada função como foi garantido o encapsulamento de dados (cópia, criação de nova estrutura, etc.);

☑ Estruturação do programa principal, designadamente main() e funções auxiliares;

☑ IU e comentários sobre decisões de navegação sobre estruturas de resultados mais complexas;

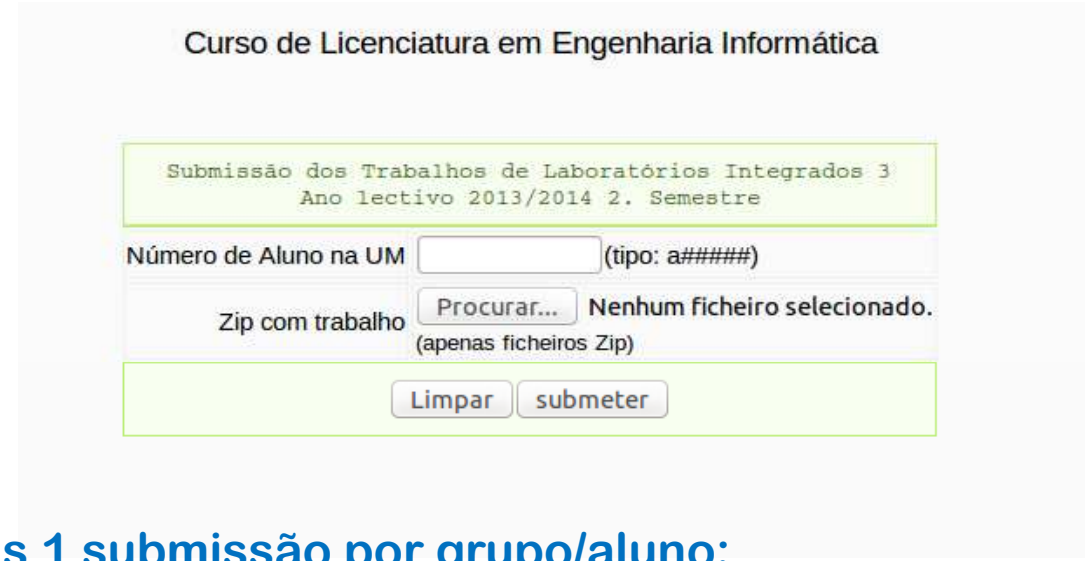
☑ Resultados gráficos e comentários sobre os testes de performance;

☑ Apresentação da makefile e do grafo de dependências;

☑ Conclusão.

Projecto de C: Submissão

☑ http://www.di.uminho.pt/submissoes



☑ Apenas 1 submissão por grupo/aluno;

☑ Pasta em formato zip contendo a makefile e todos os ficheiros .h e .c e com nome standard cf. LI3_Grupo# (cf, nº do BB);

☑ Será avaliado apenas o que for entregue em tal pasta;

☑ Datas: De 6ª. Feira, 24 de Abril às 12H00 até sábado, 26 de Abril às 23H59m.

Projecto de C: Avaliação

- Grupos inscritos no BB (+ melhorias ??);
- Não quero sequer referir o uso de JPLAG (detector de plágios);
- Apresentações: em salas e calendário a anunciar via BB);
- Cada grupo será avaliado durante +- 30 minutos;
- Podem surgir questões individuais para alguns elementos do grupo;
- A avaliação será feita com base numa grelha que reflecte todas as questões que foram pragmaticamente solicitadas relativamente ao TP de C; Assim, a avaliação será independente do docente que a realizar; Será justa, qualitativa e pragmática.