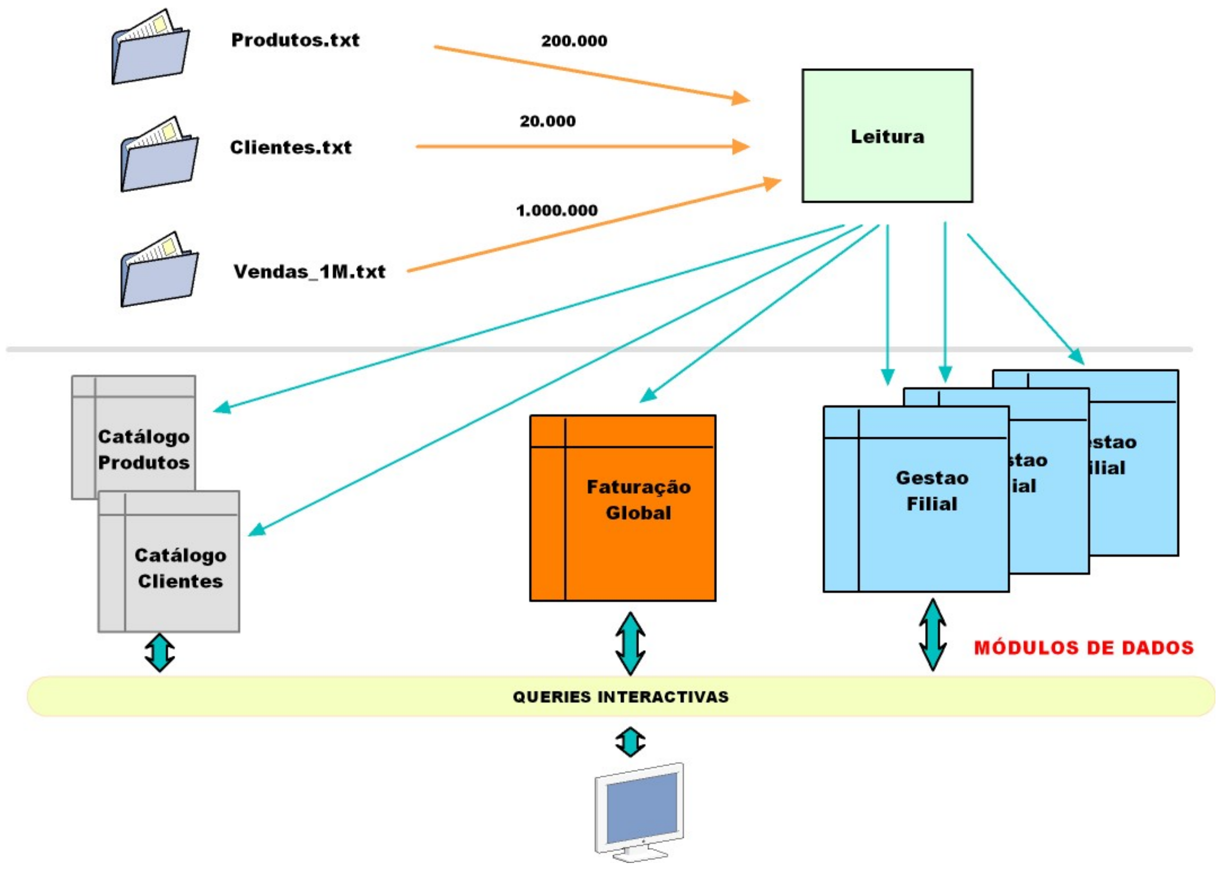
Boletim de Requisitos de Projecto Nº 1

Projecto Sistema de Gestão de Vendas - SGV



BRP Nº 1

FASE 1 = 2/3 aulas

☑ Leitura de ficheiros e validação de dados;

☑ Profiling para dimensionamento do BUFFER de fgets();

☑ Tratamento das strings lidas usando fgets();

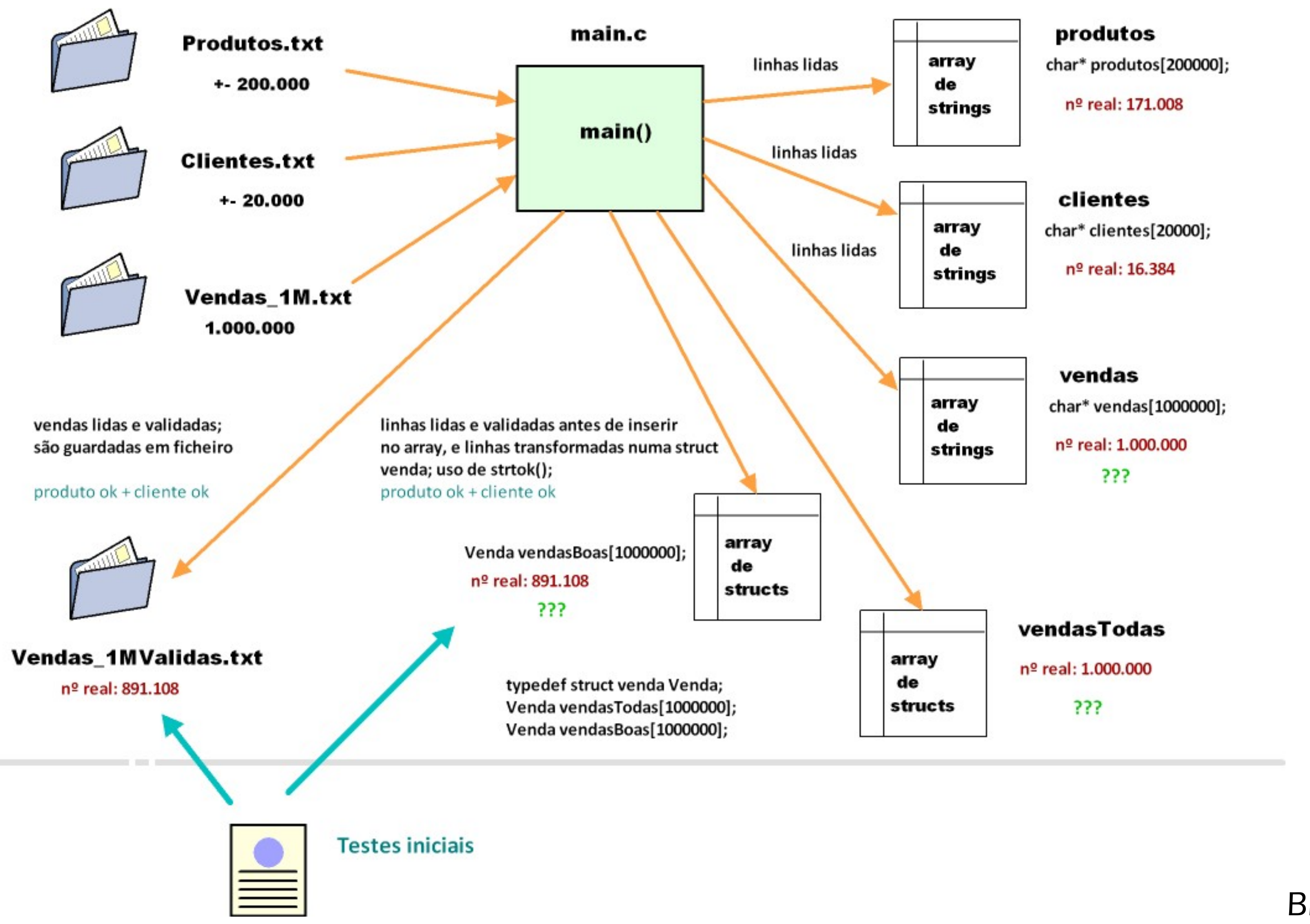
☑ Parsing de uma linha do ficheiro Vendas e validação (cf. strtok());

☑ Armazenamento das strings válidas de vendas em array de strings;

☑ Criação de um ficheiro de texto com todas as vendas válidas;

☑ Contagens finais de dados validados e guardados em arrays ;

FASE 1


Produtos.txt
+- 200.000
Clientes.txt
+- 20.000
Vendas_1M.txt
1.000.000
main.c
main()
linhas lidas
linhas lidas
linhas lidas
array de strings
produtos
char* produtos[200000];
nº real: 171.008
array de strings
clientes
char* clientes[20000];
nº real: 16.384
array de strings
vendas
char* vendas[1000000];
nº real: 1.000.000
???
vendas lidas e validadas; são guardadas em ficheiro
produto ok + cliente ok
linhas lidas e validadas antes de inserir no array, e linhas transformadas numa struct venda; uso de strtok();
produto ok + cliente ok
Venda vendasBoas[1000000];
nº real: 891.108
???
array de structs
Vendas_1MValidas.txt
nº real: 891.108
typedef struct venda Venda;
Venda vendasTodas[1000000];
Venda vendasBoas[1000000];
array de structs
vendasTodas
nº real: 1.000.000
???
Testes iniciais

FASE 1: Ler e testar valores

FIM DA LEITURA DE VENDAS_1M

Linha mais longa (Venda): ???

Produtos envolvidos: ....

Clientes envolvidos: ....

Vendas efectivas (válidas): ....

Ultimo Cliente: ....

Numero de Vendas Registadas para este cliente: ??

Numero de Vendas na Filial 1: ???

Numero de Vendas na Filial 2: ???

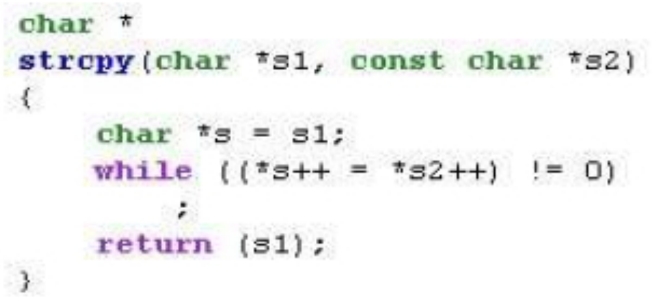
Numero de Clientes com codigo começado por A, B, ...: ???

Facturação Total registada: ???

FASE 1 **Notas pedagógicas**

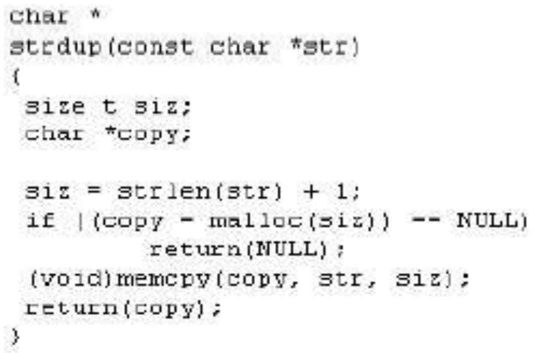
Importância de ter conhecimentos seguros sobre coisas básicas;

- 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)



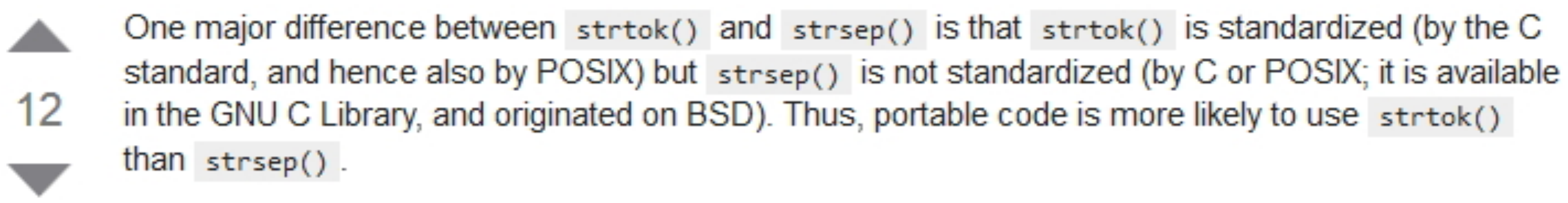
```
char *
strdup(const char *str)
{
 size t siz;
 char *copy;

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

(Aloca espaço e copia)

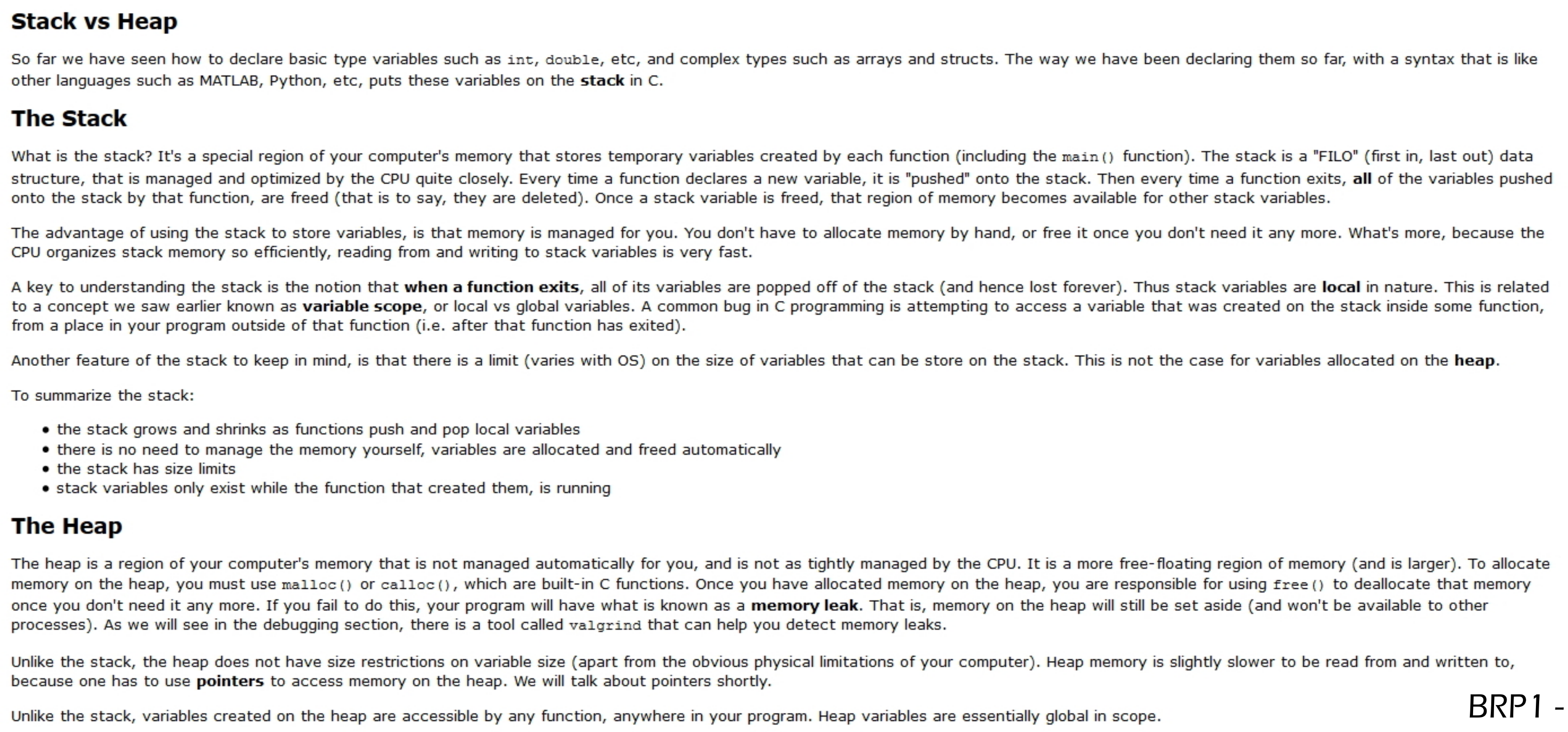
- Atenção: malloc() não inicializa; memset() não aloca; calloc() = malloc() + memset() ;
- Má compreensão destes mecanismos => segmentation faults;
- **Não descurar nunca o uso de** memcopy() **e** realloc(); **Nunca usar** bcopy();

✍ 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

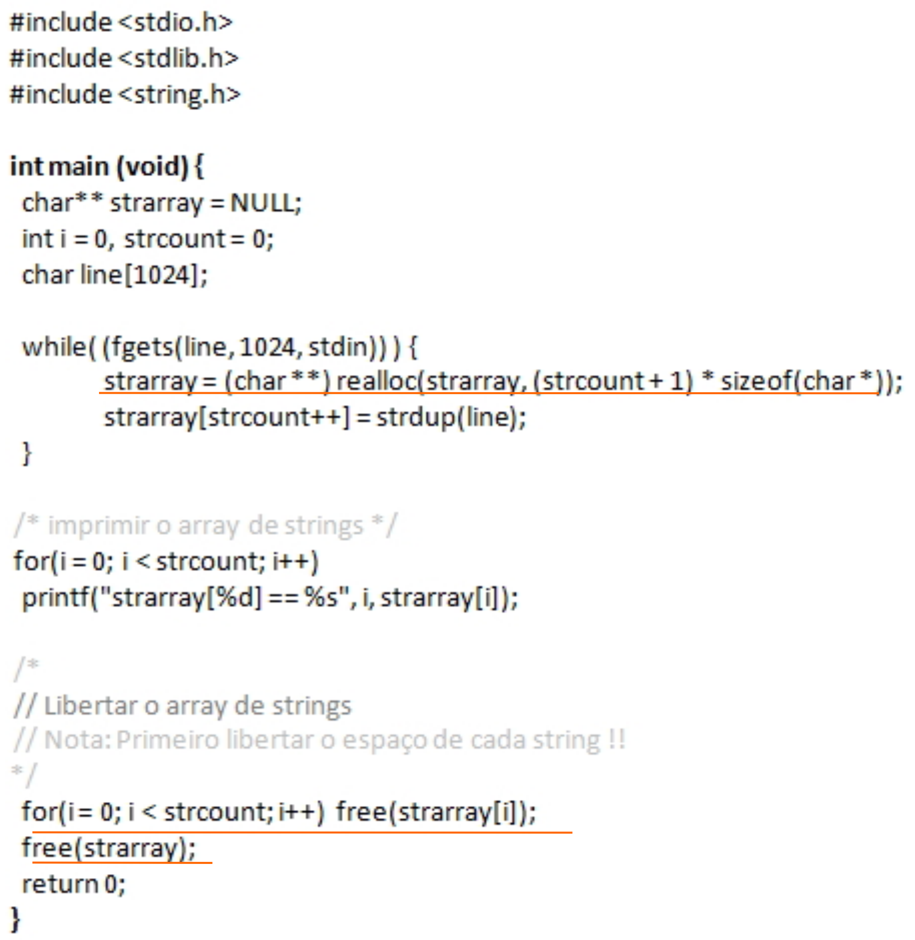
The Heap

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.

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)