# Instituto Politécnico do Cávado e Ave



# Trabalho Prático 1

# Relatório

# Jorge Miguel Arezes Noro | 15705

# Engenharia e Desenvolvimento de Jogos Digitais

# Paradigmas de Programação II

# Docente: Luís Gonzaga Martins Ferreira

2º Semestre

1. Introdução

Este documento serve de relatório ao Trabalho Prático 1 da unidade curricular de Paradigmas de Programação II. É acompanhado de um projeto de Visual Studio (*pp2-tp1.sln*) dividido em 2 projetos: *Part1 – Static Data Structures* e *Part2 – Dynamic Data Structures*. Não existem dependências entre os dois projectos.

O projecto está disponível no GitHub, no seguinte URL:   
<https://github.com/mrMav/pp2-tp1>

Na realização deste trabalho, houve como ideal apelar à parametrização do programa. Existem várias variáveis que podem ser ajustadas conforme as preferências do utilizador. Ficam abaixo ditas variáveis com a descrição do seu papel no programa:

|  |
| --- |
|  |
| Code Snippet 1- Variáveis para parametrização do programa |

1. Estruturas de Dados Estáticas
   1. Alocação de memória

Para o desenvolvimento da parte 1 do trabalho prático, a estrutura de dados estática capaz de armazenar todos os dados do cartão matriz será um *array* de inteiros. Uma possível solução, a que foi implementada neste trabalho, passou por separar as células do cartão matriz em *structs* chamados de *Nodes.* Cada *Node* terá um *array* de inteiros, em que será armazenada a sequência de números.

Para construir a matriz, usamos então outro *struct* chamado de *Row.* Este *struct* armazena o número de *Nodes* igual ao número de colunas definidas na matriz (ver *Types.h)*. O *struct* *Matrix* tem como membro um *array* do tipo *Row*, em que armazena o número de linhas definidas na matriz. É possível ver na imagem abaixo a relação dos *structs* entre si:

|  |
| --- |
|  |
| Figura 1-Relação das estruturas de dados. |

A solução pode parecer complexa demais para a resolução deste problema simples, mas desta forma deixamos espaço para o crescimento da aplicação, como por exemplo, adicionar informação sobre determinado *Node* ou sobre determinada *Row*. Uma solução mais simples e básica podia ser implementar simplesmente um *array* tridimensional:

|  |
| --- |
|  |
| Code Snippet 2- Possível implementação da matriz, usando um simples array tridimensional |

As funções de alocação de memória destes objetos são encontradas no ficheiro *ObjectCreation.c*. Fica abaixo um exemplo (alocação em memória de uma matriz):

|  |
| --- |
|  |
| Code Snippet 3- Alocação em memória de uma matriz. |

O código chama a função *createRow()*, que por sua vez chama a função *createNode()*. Esta, vai popular o *Node* com uma sequência aleatória de números. Podemos ver essa função abaixo:

|  |
| --- |
|  |
| Code Snippet 4- Alocação em memória de um Node |

* 1. Algoritmos

No que diz respeito a algoritmos, esta primeira parte do trabalho é relativamente simples. Podemos analisar como exemplo, a função em que pedimos ao utilizador que valide uma operação. Fica abaixo o código em excerto (algumas partes foram retiradas para auxiliar a leitura neste relatório, ver ficheiro *UtilityFunctions.c* para analisar o código completo):

|  |
| --- |
|  |
| Code Snippet 5- Função para validar uma operação |

A função gera números aleatórios para a linha, coluna e posição do dígito pretendido. De seguida, mostra ao utilizador qual a célula e posição do dígito que se pretende e obtém a resposta. É chamada uma função que valida essa resposta. Vejamos abaixo:

|  |
| --- |
|  |
| Code Snippet 6- Função para validar escolha do utilizador |

1. Estruturas de Dados Dinâmicas
   1. Alocação de memória

Para a parte 2 do trabalho, optar-se-á por construir uma lista ligada. Esta lista irá ser composta por *Nodes*, em que cada *Node* representa uma célula do cartão matriz e tem como dados um *array* de inteiros (os números correspondentes a essa célula). A implementação do cartão matriz como lista ligada força o acesso às células de uma forma mais cuidadosa do que com uma estrutura de dados estática. A figura abaixo mostra a estrutura de um *Node* a implementar no código:

|  |
| --- |
| C:\Users\jnoro\Source\Repos\pp2-tp1\uml-dynamic.png |
| Figura 2- Estrutura de um Node da lista ligada |

Para criar uma matriz, usa-se a função *buildMatrix()*, a qual retorna o *Node* inicial da matriz. Como não se está a usar um *array,* a função é simplesmente um ciclo que inicializa o número necessário de *Nodes.* A diferenciação de colunas e linhas da matriz, só será tida em conta nas funções de pesquisa e manipulação da matriz.Abaixo pode-se observar e analisar a implementação da função (ver *Matrix.c*) e a definição do tipo *Node* (ver *Types.c*):

|  |
| --- |
|  |
| Code Snippet 7-Alocação de memória de uma matriz, usando uma lista ligada. |

A função *addNode(Node\* head, Node\* toAdd)* (ver *CreationFunctions.c*)*,* é a responsável por alocar a memória de cada *Node* individualmente e contém dois parâmetros do tipo *Node\**. *head* é o apontador para o *Node* inicial e *toAdd* é o apontador para o *Node* a ser inserido, que no caso de ser igual a *NULL*, é criado um novo *Node* com a sequência iniciada aleatoriamente. Esta função *addNode* é também utilizada quando o utilizador carrega os dados da matriz através de um ficheiro previamente guardado. Abaixo podemos analisar a função:

|  |
| --- |
|  |
| Code Snippet 8- Inserção de um Node no final da lista |

Para criar uma novo *Node* utiliza-se uma função *createNode()* bastante similar à usada no capitulo 1, Estruturas de Dados Estáticas. Pode-se ver essa função no ficheiro *CreationFunctions.c*.

* 1. Algoritmos

Nesta secção vamos analisar alguns algoritmos interessantes necessários para a manipulação de listas ligadas. Em primeiro, observemos como foi implementada uma função que realiza uma acção em cada *Node*. Embora não fosse realmente necessário, a implementação desta função foi um desafio pessoal. O objectivo era ter uma função similar ao que outras linguagens possuem, um *forEach()*.

É então implementada a função *forEachNode(Node\* head, void(\*callback)(Node\* n).* A quem não esteja habituado, esta assinatura pode ser confusa. O primeiro parâmetro não é nada mais que o primeiro *Node* da lista. O segundo parâmetro é o apontador para uma outra função do tipo *void functionName(Node\*)*. Ao saber que em C, todos os nomes das funções são apontadores para essas mesmas funções, o segundo parâmetro torna-se um pouco mais claro, pois sabemos que um asterisco antecedente de um apontador, desreferencia e retorna o valor apontado. A explicação pode suscitar dúvidas, portanto vamos analisar a implementação da função:

|  |
| --- |
|  |
| Code Snippet 9- Implementação da função forEachNode |

No corpo da função podemos observar um algoritmo que permite percorrer a lista ligada desde o *Node head* até ao final da lista (o final da lista é definido pelo *Node* em que a propriedade *next* devolve *NULL*). Dentro do ciclo, é invocada a função que é passada como argumento e é dado como argumento o *Node* que está no momento armazenado na variável *n*. De seguida, a variável *n* obtém o valor do próximo *Node* e assim sucessivamente.

Esta abordagem permite um acesso rápido a todos os *Nodes,* como podemos ver num exemplo de utilização em que pretendemos imprimir todos os *Nodes* no ecrã. Em primeiro lugar, definimos uma função que imprima um *Node*, sem nunca esquecer qual a assinatura modelo que temos de seguir (retornar *void* e ter um parâmetro *Node\**):

|  |
| --- |
|  |
| Code Snippet 10-Função capaz de imprimir um Node no ecrã |

Assim, podemos invocar a função *forEachNode* da seguinte forma:

|  |
| --- |
|  |
| Code Snippet 11- Invocação da função forEachNode com callback |

O próximo algoritmo que será analisado, é o implementado na função *validatePosition*. O algoritmo percorre a lista ligada a partir do *Node* passado como argumento à função (será sempre, em teoria, o inicio da matriz). Para determinar em que linha e coluna nos encontramos usamos um contador que armazena em que número de *Node* estamos. Esse contador determina a coluna ao realizar a operação aritmética módulo entre si e o número de colunas na matriz e guarda o esse resultado numa variável *cCol* *(current column)*. De igual modo determina a linha da matriz em que nos encontramos, mas em vez de usar a operação módulo, usa a divisão (entre inteiros).[[1]](#footnote-1) Vejamos a implementação da função:

|  |
| --- |
|  |
| Code Snippet 12- Obter linha e coluna aritmeticamente, validatePosition em listas ligadas |

O último algoritmo que vamos rever é o implementado na função *freeMatrix(Node\*\* head)*. Esta função é a responsável de devolver ao sistema operativo a memoria requisitada através das funções *malloc()*. A função tem como parâmetro um apontador para um apontador. Isto é necessário pois após a utilização da função de C *free(void\* ptr)*, iremos alterar o valor do apontador para *NULL*. Vejamos:

|  |
| --- |
|  |
| Code Snippet 13- Função para devolver memória da matriz ao sistema operativo. |

Após verificarmos se o *Node* não é já igual a *NULL*, percorremos a árvore da mesma forma que previamente. Guardamos num *buffer* o apontador do *Node* seguinte e chamamos a função *free()* no actual *Node*. Após libertar a memória contida nesse apontador, atribuímos o seguinte *Node* ao actual e assim sucessivamente. No final, definimos o apontador inicial como *NULL*.

Após a análise destes algoritmos e para não alongar este relatório, deixam-se de parte os algoritmos para guardar a lista em ficheiro e recuperar a mesma. O algoritmo será o mesmo, pois o *core* da implementação de listas ligadas é o algoritmo para percorrer essa lista.

1. Esta forma de obter o número de linha e coluna é muito utilizada em desenvolvimento de jogos *tilebased*, em que a informação é guardada em *arrays* unidimensionais (vetores). [↑](#footnote-ref-1)