ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Projeto de Laboratório de Programação

Licenciatura em Engenharia Informática

2020/2021

Grupo 31

(8180378), Hugo Daniel Martins Silva

(8200690), Diogo Xavier Fonseca Vieira

Estrutura do Documento

Este documento descreve todo o trabalho feito durante a realização do projeto de Laboratórios de Programação. Ao longo do documento será apresentado uma descrição do projeto realizado e das decisões tomadas no decorrer do seu desenvolvimento. Serão também apresentadas as funcionalidades requeridas, as estruturas de dados e a codificação implementada para resolução destas. Seguido das funcionalidades requeridas serão apresentadas as funcionalidades propostas pelo grupo, bem como o interesse que estas apresentam para a empresa.

Demonstradas as funcionalidades será apresentada a estrutura analítica do projeto em que é abordada a divisão do projeto em tarefas e sub-tarefas. Seguido a estrutura é apresentado as funcionalidades implementadas face às funcionalidades propostas.

Por fim é apresentado a conclusão do grupo acerca da realização do projeto.

1. Introdução

O projeto desenvolvido consiste num programa que permite processar salários de funcionários através da importação de um ficheiro de texto. Para o processamento de salário são importadas as tabelas de Segurança Social e IRS. As tabelas podem ser alteradas sendo que o utilizador pode editar os critérios existentes, adicionar novos ou eliminar. Feito o processamento de salário o utilizador pode também gerar um relatório em ficheiro de texto onde é guardada toda a informação relativa ao processamento (ex: salário bruto, líquido e subsídio de alimentação);

O programa permite também gestão de funcionários, ou seja, o utilizador pode criar funcionários, editar as suas informações e pode também removê-los. Toda a informação sobre os funcionários pode também ser guardada em ficheiro, caso o utilizador assim o pretenda.

Por fim o programa permite também exibir listagens que são do interesse da empresa. Entre as listagens é possível ao utilizador verificar os custos totais que a empresa teve num determinado mês. É também possível saber a percentagem de funcionários por idades.

No decorrer do projeto foram tomadas algumas decisões para a resolução do problema. Entre as opções tomadas está a decisão de colocar as três tabelas de IRS num só ficheiro. Com esta opção foi adicionado ao ficheiro uma coluna extra que identifica a tabela a que um critério corresponde. Por exemplo uma linha na qual o estado civil seja 1 significa que é uma linha que corresponde à tabela “NÃO CASADO”.

No ficheiro da tabela de Segurança Social foi definido na descrição dos critérios que os espaços seriam substituídos por “_” como mostra o seguinte exemplo.

Descrição de um critério:

“Trabalhadores em geral”

Descrição armazenada no ficheiro

"Trabalhadores_em_geral"

Esta decisão foi tomada devido à função que foi utilizada para ler o ficheiro, scanf[1]. O scanf quando formatado para ler uma String lê até que um espaço em branco seja encontrado. Assim, e como o objetivo era que toda a descrição fosse armazenada numa *String,* foram substituídos os espaços em branco pelo "_".

A estrutura funcionário, figura1, contém também um critério do tipo *int* que identifica a taxa de segurança social a aplicar nos cálculos. O motivo para esta decisão foi o facto de o utilizador conseguir adicionar novas taxas contributivas e identificar os funcionários que estão sujeitos a ela.

Exemplo: Utilizador cria uma taxa para administradores com mais de 60 anos e o um dependente.

Para que a taxa seja aplicada num funcionário é necessário alterar o seu critério de segurança social (colocando o número da nova taxa).

Com esta implementação o utilizador pode criar as taxas que pretender, sendo que tem de atualizar um atributo no funcionário para estas serem aplicadas nos cálculos.

Outra das decisões tomadas no decorrer do projeto foi armazenar a lista de funcionários num ficheiro binário, em vez de um ficheiro de texto. Esta decisão foi tomada pelo facto de um ficheiro binário ser mais indicado para guardar estruturas de dados. Um ficheiro binário normalmente contém também tempos de acesso mais reduzidos que um ficheiro de texto[2].

[1] Função da linguagem c que permite a leitura de dados a partir de uma fonte caracteres de acordo com um formato pré-definido.

[2] Informação obtida dos slides teóricos da disciplina de Fundamentos de Programação

2. Funcionalidades requeridas

2.1 Gestão de Funcionários

A gestão de funcionários solicitava duas estruturas de dados. Uma estrutura que armazenasse toda a informação relativa a um funcionário, e uma estrutura que permitisse armazenar uma lista de funcionários.

Para gestão de funcionários eram necessárias as seguintes funções CRUD:

Create- criação de um novo funcionário, sendo que para ser adicionado com sucesso à lista de funcionários teria de ter um código que ainda não existisse, uma vez que o código é único.

Read-listar as informações de um funcionário através do seu código.

Update- atualizar as informações de um funcionário, caso ele exista na lista.

Delete- colocar a informação no funcionário como "Removido".

Para a correta realização das funções de gestão de funcionários foram também criadas algumas funções adicionais como por exemplo a função de verificar se um código existe na lista e a função de expandir memória.

A função que verifica se um código existe na lista é utilizada em todas funções de gestão. Já a função de expandir memória é utilizada quando o limite de memória alocada é atingido, sendo que quando isso sucede é feita uma nova realocação de memória.

Estruturas criadas para gestão de funcionários:

```
typedef struct {
    int codigo; //Unico
    char nome[TAMANHO_STRING];
    char numeroTelemovel[TAMANHO_TELEMOVEL];
    Data dataNascimento;
    Data dataEntradaEmpresa;
    Data dataSaidaEmpresa;
    EstadoCivil estadoCivil;
    EstadoFuncionario estadoEmpregado;
    Cargo  cargoEmpresa;
    int nrDependentes;
    float subsidioAlimentacao;
    float valorHora;
    int criterioSS;
} Funcionario;
```

Figura 1 Estrutura Funcionário

A estrutura "Funcionario" (figura1) contém dois valores do tipo *float* que armazenam o valor/hora que o funcionário recebe e o valor do subsídio de alimentação. Contém três inteiros que guardam o código, o número de dependentes e a taxa a aplicar da segurança social. Contém três enumerações, uma relativa ao cargo, outra relativa ao estado do funcionário na empresa e outra relativa ao estado civil. Por fim a estrutura possui dois arrays de caracteres para guardar o nome do funcionário e o número de telemóvel.

```
typedef struct {
    int nrFuncionarios;
    int tamanho;
    Funcionario *listafuncionarios;
} ListaFuncionarios;
```

Figura 2 Estrutura Lista de Funcionários

A estrutura "ListaFuncionarios" (figura2) contém um apontador para a estrutura "Funcionario" que irá servir para alocar memória. Contém também dois valores inteiros, um corresponde ao tamanho da lista e outro ao número de funcionários presentes na lista.

Funções utilizadas na gestão de funcionários.

```c
int adicionarFuncionario(ListaFuncionarios *listafuncionarios) {
    Funcionario func;
    criarFuncionario(&func);
    //lista cheia
    if (listafuncionarios->nrFuncionarios == listafuncionarios->tamanho) {
        expandirMemoria(listafuncionarios);
    }
    //verifica se  código já existe
    for (int i = 0; i < listafuncionarios->nrFuncionarios; i++) {
        if (listafuncionarios->listafuncionarios[i].codigo == func.codigo) {
            return 0;
        }
    }
    //Possivel a insercao, adiciona e incrimenta o nr de funcionários
    listafuncionarios->listafuncionarios[listafuncionarios->nrFuncionarios] = func;
    listafuncionarios->nrFuncionarios++;
    return 1;
}
```

Figura 3 Função Adicionar Funcionários à lista

A função "adicionarFuncionario" (figura3) permite a inserção de um funcionário na lista, esta lê um funcionário da consola, depois verifica se o código inserido pelo utilizador já existe na lista. Caso exista retorna 0 e a inserção é cancelada. Em caso de não existir, verifica se é necessário realocar memória para a inserção do funcionário, se for necessário faz a realocação, e de seguida insere o funcionário na lista, incrementando o número de funcionários.

```c
void mostraFuncionario(ListaFuncionarios *listafuncionarios, int idFunc) {
    Funcionario *fun;
    fun = obterFuncionario(listafuncionarios, idFunc);
    if (fun != NULL) {
       mostrarDadosFuncionario(*fun);
    } else {
        puts(MSG_NAO_ENCONTRADO);
    }
}
```

Figura 4 Função que mostra dados de um funcionário

A função "mostraFuncionario" (figura4) mostra os dados de um funcionário, caso o código passado como parâmetro exista na lista de funcionários.

```
int obterIndiceFuncionario(ListaFuncionarios *listafuncionarios, int idFunc) {
    int i = 0;

    while (i < listafuncionarios->nrFuncionarios && listafuncionarios->listafuncionarios[i].codigo != idFunc){
        i++;
    }
    if (i == listafuncionarios->nrFuncionarios) { //Chegou ao fim da lista não encontrou
        return -1;
    }
   return i;
}
```

Figura 5 Função que devolve posição de funcionário na lista

A função "obterIndiceFuncionario" (figura5) percorre toda a lista de funcionários e verifica se um código existe. Caso exista devolve a posição em que se encontra, se não existir devolve -1.

```
void apagarFuncionario(ListaFuncionarios *listafuncionarios, int idFunc) {
    Funcionario *funcionario = obterFuncionario(listafuncionarios, idFunc);
    if (funcionario != NULL) {
        //ESTADO passa REMOVIDO,
        funcionario->estadoEmpregado = 0;
        puts(MSG_REMOVIDO_SUCESS);
    } else {
        puts(MSG_NAO_ENCONTRADO);
    }
}
```

Figura 6 Função remove funcionário

A função "apagarFuncionario" (figura6) recebe um código como parâmetro e verifica se ele existe na lista, caso exista coloca o estado do funcionário a "Removido". Em caso de não existir informa.

```
void atualizaFuncionario(ListaFuncionarios *listafuncionarios, int idFunc) {
    Funcionario *fun;
    fun = obterFuncionario(listafuncionarios, idFunc);
    if (fun != NULL) {
        menuEditarFuncionario(fun);
    } else {
        puts(MSG_NAO_ENCONTRADO);
    }
}
```

Figura 7 Função atualiza funcionário

A função "atualizaFuncionario" (figura7) recebe um código como parâmetro e verifica se ele existe na lista, caso exista mostra um menu onde o utilizador pode escolher as informações que pretende alterar. Em caso de o código não existir, informa.

```
void expandirMemoria(ListaFuncionarios *listafuncionarios){
    Funcionario *func = (Funcionario*) realloc(listafuncionarios->listafuncionarios, sizeof(Funcionario) * (listafuncionarios->tamanho * 2));
    listafuncionarios->listafuncionarios= func;
    listafuncionarios->tamanho *= 2;

}
```

Figura 8 Função que expande Memória da lista Funcionários

A função "expandirMemoria" (figura 8) é invocada quando o número de funcionários é igual ao tamanho de memória alocada, esta função faz a realocação de memória, duplicando o seu tamanho.

Lista Funcionários Leitura/Escrita em Ficheiro

A escrita da lista de funcionários em ficheiro foi, como referido na introdução deste documento, realizada para um ficheiro binário. Para a escrita foi criada uma função que abre uma *Stream*[3] de *write* para o ficheiro e escreve o conteúdo da lista.

```
void guardarFuncionariosFx(ListaFuncionarios *listafuncionarios,char *nomeficheiro){
    FILE *fp = fopen(nomeficheiro, "w");
    if (fp != NULL) {
        fwrite(&listafuncionarios->nrFuncionarios, sizeof (int),1, fp);
        fwrite(listafuncionarios->listafuncionarios, sizeof (Funcionario), listafuncionarios->nrFuncionarios, fp);
        fclose(fp);
    }else{
        puts(MSG_ERRO_FILE);
    }
}
```

Figura 9 Função guardar lista Funcionários

A figura 9 mostra a função que guarda a lista de funcionários em ficheiro, esta função apenas é invocada caso a lista tenha pelo menos um funcionário. No caso de existir alguma informação já presente no ficheiro é substituída. No ficheiro é armazenado o número de funcionários da lista e de seguida os funcionários.

Para a leitura da lista de funcionários, a primeira tarefa executada é a libertação de memória alocada, este procedimento é realizado porque a leitura de ficheiro implica a substituição de toda a memória central do computador. Depois de libertada a memória é realizada uma *Stream* de leitura ao ficheiro que contém a lista.

[3] Uma *stream* é uma fonte de input ou destino de output.

```
void carregarFuncionariosFx(ListaFuncionarios *listafuncionarios,char *nomeficheiro){
    int sucesso = 0;
    FILE *fp = fopen(nomeficheiro, "rb");
    if (fp != NULL) {
        fread(&listafuncionarios->nrFuncionarios, sizeof (int), 1, fp);
        if(listafuncionarios->nrFuncionarios > 0){
         listafuncionarios->listafuncionarios = (Funcionario*) malloc(listafuncionarios->nrFuncionarios * sizeof(Funcionario));
         fread(listafuncionarios->listafuncionarios, sizeof (Funcionario), listafuncionarios->nrFuncionarios, fp);
         listafuncionarios->tamanho=listafuncionarios->nrFuncionarios;
          sucesso = 1;
        }
       fclose(fp);
    }
    if(!sucesso){
        fp = fopen(nomeficheiro, "wb");
         listafuncionarios->listafuncionarios =(Funcionario *) malloc(TAMANHO_MAX *sizeof(Funcionario));
         listafuncionarios->tamanho = TAMANHO_MAX;
         listafuncionarios->nrFuncionarios=0;
         fclose(fp);
    }
}
```

Figura 10 Função de leitura da lista de funcionários

A função "carregarFuncionariosFX", figura 10, inicia uma *Stream* de leitura. Caso a leitura seja válida lê o número de funcionários da lista. De seguida aloca a memória necessária para carregar os funcionários e faz a respetiva leitura. Caso a leitura não tenha sido feita, coloca o número de funcionários a 0 e aloca uma quantidade de memória predefinida.

2.2 Gestão Tabela Segurança Social

Importação da tabela do ficheiro de texto

A primeira tarefa realizada foi a importação da tabela de segurança social através de um ficheiro de texto. Para esta tarefa foram criadas duas estruturas.

```
typedef struct  {
    char  descTaxa[MAX_DESCRICAO_TABELA];
    float EntidadeEmpregadadora;
    float Trabalhador;
    float Global;
    int identicacaoCriterio;
} LinhasTabela;
```

Figura 11 Estrutura das linhas da tabela da Segurança Social

A primeira estrutura (figura11) foi criada para armazenar as linhas da tabela, contém três valores *float* para guardar as percentagens das contribuições, contém um valor *int* que identifica a taxa e por fim contém um *array* de caracteres que armazena a descrição da taxa.

```c
typedef struct  {
    char  descricaoColuna0[MAX_DESCRICAO_TABELA];
    char  descricaoColuna1[MAX_DESCRICAO_TABELA];
    char  descricaoColuna2[MAX_DESCRICAO_TABELA];
    char  descricaoColuna3[MAX_DESCRICAO_TABELA];
    int numeroCriterios;
    int tamanhoCriterios;
    Criterios *criterios ;
} TabelaSegurancaSocial;
```

Figura 12 Estrutura que armazena tabela de segurança social

Com a estrutura para armazenar as linhas criada, foi adicionada a estrutura para armazenar a tabela. A estrutura contém 4 arrays de caracteres para guardar os nomes das colunas, contém também um valor *int* para armazenar o número de linhas da tabela e um apontador para a estrutura "LinhasTabela" apresentada na figura 11, utilizada para alocar a memória.

Com as estruturas criadas, foi desenvolvida uma função que lê a tabela do ficheiro e a armazena em memória.

```c
void lerFicheiroSegurancaSocial(TabelaSegurancaSocial *lista, char*filename) {
    int i = 0, w = 0, sucesso = 0;
    lista->numeroLinhas = numeroLinhas(filename);
    LinhasTabela crite;
    FILE *file;
    file = fopen(filename, "r");
    if (lista->numeroLinhas > 0) {
        lista->criterios = (LinhasTabela*) malloc(sizeof (LinhasTabela) * lista->numeroLinhas);
        fscanf(file, "%s %s %s %s", lista->descricaoColuna0, lista->descricaoColuna1, lista->descricaoColuna2, lista->descricaoColuna3);
        while (i < lista->numeroLinhas) {
            fscanf(file, "%d,%s %f,%f,%f", &crite.identicacao, &crite.tipoTrabalhador, &crite.EntidadeEmpregadadora, &crite.Trabalhador, &crite.
            lista->criterios[i] = crite;
            i++;
        }
        sucesso = 1;
        lista->tamanhoCriterios = lista->numeroLinhas;
        fclose(file);
        //Remove _ da discrição
        for (int j = 0; j < lista->numeroLinhas; j++) {
            w = 0;
            while (lista->criterios[j].tipoTrabalhador[w] != '\0') {
                if (lista->criterios[j].tipoTrabalhador[w] == '_') {
                    lista->criterios[j].tipoTrabalhador[w] = ' ';
                }
                w++;
            }
        }
    }
```

Figura 13 Função que importa a tabela de segurança social

A função, representada na figura 13, abre uma *Stream* de leitura no ficheiro. Caso a leitura seja bem-sucedida guarda os valores na estrutura "TabelaSegurancaSocial". A primeira operação executada é a obtenção do número de linhas que o ficheiro possui através da função "numeroLinhas". Esta operação é executada para obter o tamanho de

memória que é necessário alocar para as linhas da tabela, que são armazenadas na estrutura da figura 11. Depois de alocada a memória são então lidos os valores do ficheiro e armazenados na estrutura que representa a tabela. Caso a leitura do ficheiro não seja feita corretamente não é alocada memória, uma vez que, para realizar operações é necessário que a importação seja realizada corretamente.

Edição da Tabela

A edição permite realizar três operações, editar, remover e adicionar linhas. Para adicionar uma nova linha, é pedido ao utilizador que insira os dados. De seguida é verificado se é necessário realocar a memória para guardar a nova linha. Caso seja, é invocada função que executa a realocação de memória, como mostra a figura 14.

```
void adicionarNovosCriteriosSS(TabelaSegurancaSocial *lista) {
    LinhasTabela criterio;
    int w = 0;
    if (lista->numeroLinhas == lista->tamanhoCriterios) {
        expandirMemoriaTabelaSS(lista);
    }
    puts("Insira a descrição númerica da Tabela  [ex 1, 2,]");
    scanf("%d", &criterio.identicacao);
    cleanInputBuffer();
    puts("Escreva a descrição do criterio");
    fgets(criterio.tipoTrabalhador, MAX_DESCRICAO_TABELA, stdin);
    puts("Escreva a percentagem da Entidade Empregadora");
    scanf("%f", &criterio.EntidadeEmpregadadora);
    puts("Insira nova percentagem do Funcionario");
    scanf("%f", &criterio.Trabalhador);
    criterio.Global = criterio.EntidadeEmpregadadora + criterio.Trabalhador;
    //Retirar os espaços da descriçao
    while (criterio.tipoTrabalhador[w] != '\0') {
        if (criterio.tipoTrabalhador[w] == ' ') {
            criterio.tipoTrabalhador[w] = '_';
        }
        w++;
    }

    lista->criterios[lista->numeroLinhas] = criterio;
    lista->numeroLinhas++;
}
```

Figura 14 Adicionar novos critérios à tabela

Feita a realocação, no caso de ter sido necessário, é adicionada a nova linha à tabela e incrementado o número de linhas que esta possui.

Para efetuar a remoção de uma linha é pedido ao utilizador que insira a linha que pretende eliminar, depois é verificado se esta corresponde a uma linha que exista. No caso de não existir, informa e pede uma nova linha ao utilizador.

Caso a linha pertença à tabela, é removida. A função que executa esta operação é apresentada na figura 15.

```
void removerLinha(TabelaSegurancaSocial *lista) {
    int linhaRemover;
    do {
        printf("Insira a linha da tabela que pretende remover [0-%d]\n ", lista->numeroLinhas);
        scanf("%d", &linhaRemover);
        if (linhaRemover < 0 || linhaRemover > lista->numeroLinhas) {
            puts("Não existe essa linha na tabela!");
        }
    } while (linhaRemover < 0 || linhaRemover > lista->numeroLinhas);

    for(int i = linhaRemover; i < lista->numeroLinhas; i++){
        lista->criterios[i]= lista->criterios[i+1];
    }
    lista->numeroLinhas--;
    puts("Linha removida com sucesso!");
}
```

Figura 15 Função que remove linha da tabela segurança social

Para a edição de uma linha, tal como na remoção, é pedido ao utilizador que insira uma linha a editar, de seguida é verificado se esta existe na tabela. No caso de não existir, informa e pede uma nova linha ao utilizador. Caso exista na tabela é apresentado o menu de edição (figura 16).

```
Escolha o que pretende editar
Editar Tabela Segurança Social
------------------------------
1. Percentagem Entidade Empregadora
2. Percentagem Trabalhador
0. Concluir
```

Figura 16 Menu edição da linha da tabela

O utilizador escolha a opção e insere a nova percentagem, quando a percentagem é inserida o atributo que corresponde à coluna “Global” é atualizado automaticamente. A coluna “Global” corresponde à soma entre as percentagens da entidade empregadora e do trabalhador. O código da função que executa esta operação é apresentado na figura 17.

```c
void editarTabelaSS(TabelaSegurancaSocial *lista) {
    int auxx, opcao;
    do {
        printf("Insira a linha da tabela que pretende editar [0-%d]\n ", lista->numeroLinhas);
        scanf("%d", &auxx);
        if (auxx < 0 || auxx > lista->numeroLinhas) {
            puts("Não existe essa linha na tabela!");
        }
    } while (auxx < 0 || auxx > lista->numeroLinhas);

    puts("Escolha o que pretende editar");

    do {
        puts("Editar Tabela Segurança Social");
        puts("-----------------------------");
        puts("1. Percentagem Entidade Empregadora");
        puts("2. Percentagem Trabalhador");
        puts("0. Concluir");
        scanf("%d", &opcao);
        switch (opcao) {
            case 1:
                lista->criterios[auxx].EntidadeEmpregadadora = obterPercentagem();
                lista->criterios[auxx].Global = lista->criterios[auxx].EntidadeEmpregadadora + lista->criterios[auxx].Trabalhador;
                break;
            case 2:
                lista->criterios[auxx].Trabalhador = obterPercentagem();
                lista->criterios[auxx].Global = lista->criterios[auxx].EntidadeEmpregadadora + lista->criterios[auxx].Trabalhador;
                break;
        }
    } while (opcao != 0);

}
```

Figura 17 Função editar tabela segurança social

Todas as operações em cima descritas (Remover, Adicionar, Editar) são efetuas em memória, não tendo nenhuma interação com o ficheiro. Para que as alterações sejam guardadas em ficheiro, o utilizador tem de escolher essa opção no menu. Quando escolhida, é invocada a função que escreve a tabela no ficheiro, substituindo a informação existente.

```c
void escreverTabelaFicheiro(TabelaSegurancaSocial lista, char *filename) {
    //Adiciona _ em vez de espacao discrição
    int w = 0;
    for (int j = 0; j < lista.numeroLinhas; j++) {
        w = 0;
        while (lista.criterios[j].tipoTrabalhador[w] != '\0') {
            if (lista.criterios[j].tipoTrabalhador[w] == ' ') {
                lista.criterios[j].tipoTrabalhador[w] = '_';
            }
            w++;
        }
    }
    FILE *file;
    file = fopen(filename, "w");
    if (file != NULL) {
        fprintf(file, "%s %s %s %s \n", lista.descricaoColuna0, lista.descricaoColuna1, lista.descricaoColuna2, lista.descricaoColuna3);
        for (int i = 0; i < lista.numeroLinhas; i++) {
            fprintf(file, "%d,%s %.2f,%.2f,%.2f \n", lista.criterios[i].identicacao, lista.criterios[i].tipoTrabalhador, lista.criterios[i].En
        }
    } else {
        puts(MSG_ERRO_READ_FILE);
    }
fclose(file);
}
```

Figura 18 Função que escreve a tabela de segurança social no ficheiro

A função que escreve a tabela no ficheiro está representada na figura 18, esta função escreve toda a tabela no ficheiro.

2.3 Gestão Tabela IRS

Importação da tabela do ficheiro de texto

A primeira tarefa realizada foi a importação das três tabelas de IRS através de um ficheiro de texto. Para esta tarefa foram criadas duas estruturas.

```
typedef struct {
    EstadoCivil ident;
    float nivelSalario;
    float semDependentes;
    float umDependente;
    float doisDependentes;
    float tresDependentes;
    float quatroDependentes;
    float cincoMaisDependentes;
} Irs;
```

Figura 19 Estrutura que armazena uma linha da tabela de IRS

A primeira estrutura (figura19) foi criada para armazenar as linhas da tabela, contém 7 valores *float* para guardar as percentagens das taxas e o nível salarial. Contém ainda uma enumeração que identifica a tabela a que cada linha pertence (Não Casado, Casado Único titular, Casado Dois Titulares).

```
typedef struct {
    int numeroCondicoes;
    int tamanho;
    char descricaoNivelSalario[SIZE_TIPO_FUNCIONARIO];
    char descricao_sem_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_um_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_dois_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_tres_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_quatro_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_cinco_filhos[SIZE_TIPO_FUNCIONARIO];
    char descricao_estado_civil[SIZE_TIPO_FUNCIONARIO];
    Irs *listaCriterios;
} TabelaIrs;
```

Figura 20 Estrutura que armazena as tabelas de IRS

Com a estrutura para armazenar as linhas das tabelas IRS criada, foi adicionada a estrutura para armazenar a tabela. A estrutura contém 8 arrays de caracteres para

guardar os nomes das colunas, contém também um valor *int* para armazenar o número de linhas da tabela e um apontador para a estrutura "Irs" apresentada na figura 19, que servirá para alocar a memória necessária para as linhas.

Com as estruturas criadas, foi desenvolvida uma função que lê a tabela do ficheiro e a armazena em memória.

A função, abre uma *Stream* de leitura de ficheiro. Caso a leitura seja bem-sucedida guarda os valores na estrutura "TabelaIrs". A primeira operação executada é a obtenção do número de linhas que o ficheiro possui. Esta operação é executada para obter o tamanho de memória que é necessário alocar. Depois de alocada a memória são então lidos os valores do ficheiro e armazenados na estrutura que representa a tabela. Caso a leitura do ficheiro não seja feita corretamente não é alocada memória, uma vez que para as operações de gestão é necessário que a importação seja realizada.

Edição da Tabela

A edição permite realizar três operações, editar, remover e adicionar linhas. Para adicionar uma linha, é pedido ao utilizador a tabela na qual esta vai ser adicionada. De seguida é verificado se é necessário realocar a memória para guardar a nova linha. Caso seja, é invocada função que executa a realocação de memória, como mostra a figura 21.

```
/**
 *
 * @param tabelaIrs apontador para a tabela de IRS
 */
void adicionarNovosCriterios(TabelaIrs *tabelaIrs){
    int tabela,i;
    Irs irs;
    if(tabelaIrs ->numeroCondicoes == tabelaIrs->tamanho){
        expandeMemoriaTabelaIrs(tabela);
    }
    tabela = obterTabela();
    irs.ident=tabela;

    obterNovoCriterioIRS(&irs);
    for( i = 0; i < tabelaIrs ->numeroCondicoes; i++){
        if(irs.ident == tabelaIrs->listaCriterios[i].ident && irs.nivelSalario < tabelaIrs->listaCriterios[i].nivelSalario){
            break;
        }
    }
    if(i == tabelaIrs->numeroCondicoes ){
         for( i = 0; i < tabelaIrs ->numeroCondicoes; i++){
             if(tabelaIrs->listaCriterios[i].ident == irs.ident && tabelaIrs->listaCriterios[i].ident != tabelaIrs->listaCriterios[i+1].ident){
                 i++;
                 break;
             }
         }
    }
    tabelaIrs->numeroCondicoes++;
    for(int j = tabelaIrs ->numeroCondicoes; j > i; j-- ){
        tabelaIrs->listaCriterios[j]= tabelaIrs->listaCriterios[j-1];
    }
    tabelaIrs->listaCriterios[i]= irs;
}
```

Figura 21 Adição de nova linha à tabela de IRS

Depois de verificado se era necessário fazer o reajuste de memória, é pedido ao utilizador para inserir os valores que deseja, desde o nível salarial às taxas por número de dependentes.

Assim que termina a leitura dos valores é inserida a linha na tabela, respeitando a ordem crescente do nível salarial. Para isso, o nível salarial é comparado com os restantes da tabela e quando um for maior significa que é encontrada a posição onde linha tem de ser inserida. De seguida, todos os valores são colocados uma casa à frente na lista, para que a nova linha seja inserida na posição correta e a tabela continue ordenada pelo nível salarial. A função que executa a adição de uma linha é apresentada na figura 21.

Para efetuar a remoção de uma linha é pedido ao utilizador a tabela na qual esta vai ser eliminada. Depois é pedido o número da linha e verificado. No caso de não corresponder, informa e pede uma nova linha ao utilizador.

Caso a linha pertença à tabela, é removida e o número de linhas é atualizado. A função que executa esta operação é apresentada na figura 22.

```
void removerLinhaTabelaIrs(TabelaIrs *tabelaIrs){
    int linha, aux = 0;
    int tabela = obterTabela();
    switch (tabela) {
        case 1:
            imprimirTabelaNaoCasado(*tabelaIrs);
            break;
        case 2:
            imprimirTabelaUnicoTitular(*tabelaIrs);
            break;
        case 3:
            imprimirTabelaDoisTitular(*tabelaIrs);
            break;
    }
     do {
        puts("Escolha a LINHA que pretende Remover");
        scanf("%d", &linha);
        if (tabelaIrs->listaCriterios[linha].ident == tabela) {
            aux = 1;
        } else {
            puts("Linha NÃO pertence a esta tabela");
        }

    } while (aux == 0);
      tabelaIrs->numeroCondicoes--;

      for(int i = linha; i < tabelaIrs->numeroCondicoes ; i++){
        tabelaIrs->listaCriterios[i]=tabelaIrs->listaCriterios[i + 1];
    }
}
```

Figura 22 Função que remove linha tabela IRS

Para a edição de uma linha, tal como na remoção, é pedido ao utilizador a tabela na qual pretende editar. Depois é pedido o número da linha. Com o número da linha obtido é verificado se esta pertence à tabela. No caso de não pertencer, informa e pede uma nova linha ao utilizador.

Caso a linha pertença à tabela, é apresentado o menu ao utilizador com os vários campos que podem ser alterados.

```
Edição IRS
---------------------------
1. Percentagem SEM Dependentes
2. Percentagem UM Dependentes
3. Percentagem DOIS Dependentes
4. Percentagem TRES Dependentes
5. Percentagem QUATRO Dependentes
6. Percentagem CINCO OU MAIS Dependentes
7. NIVEL de Salario
0. Concluir
```

Figura 23 Menu de edição linha da tabela IRS

O utilizador pode também gravar a tabela em ficheiro (com ou sem alterações). Quando escolhida a opção gravar, é invocada a função que escreve a tabela no ficheiro, substituindo a informação existente.

```
void escreverTabelaFicheiroIRS(TabelaIrs lista_tabelas, char *filename) {
    FILE *file;
    file = fopen(filename, "w");
    int i = 0;
    if (file != NULL) {
        fprintf(file, "%s %s %s %s %s %s %s %s\n", lista_tabelas.descricao_estado_civil, lista_tabelas.descricaoNivelSalario
                lista_tabelas.descricao_um_filhos, lista_tabelas.descricao_dois_filhos,
                lista_tabelas.descricao_tres_filhos, lista_tabelas.descricao_quatro_filhos, lista_tabelas.descricao_cinco_fi

        while (i < lista_tabelas.numeroCondicoes) {
            fprintf(file, "%d,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f\n", lista_tabelas.listaCriterios[i].ident, lista_tabelas.li
                    lista_tabelas.listaCriterios[i].tresDependentes, lista_tabelas.listaCriterios[i].quatroDependentes, list
            i++;
        }

    } else {
        puts(MSG_ERRO_READ_FILE);
    }
  fclose(file);
}
```

Figura 24 Função que escreve a tabela IRS no ficheiro

A função que escreve a tabela no ficheiro está representada na figura 24, esta função escreve toda a tabela no ficheiro.

2.4 Processamento de Salário

A funcionalidade de processamento de salário solicitou cincos estruturas, duas para armazenar a informação necessária para processar salário (código funcionário, número de dias trabalhados, faltas etc.), e outras duas estruturas para guardar a informação relativa ao processamento do salário (salário líquido, salário bruto, retenção IRS, etc.). Foi também criada uma estrutura para representar todos os meses.

A informação necessária para processar salário é importada de um ficheiro de texto. As duas estruturas criadas para armazenar a informação estão apresentadas nas figuras 25 e 26 respetivamente.

```
typedef struct {
    int codigoFuncionario; //Unico
    int nrDiasSemana;
    int nrMeiosDias;
    int nrDiasFimdeSemana;
    int nrFaltas;
} DadosProcessamentoSalario;
```

Figura 25 Estrutura que armazena dados para um processamento

```
typedef struct{
    int numeroSalariosaProcessar;
    DadosProcessamentoSalario *listaProcessamentos;
} ListaProcessamentos;
```

Figura 26 Estrutura para armazenar um ficheiro com dados para processamento de salário

A figura 25 representa uma estrutura que armazena informação necessária para processar um salário. Todos os são valores inteiros.

A figura 26 armazena um ficheiro de importação. Possui o número de salários a processar e um apontador para a estrutura da figura 25. Este apontador serve para, quando o ficheiro for importado, alocar a memória necessária de acordo com o número de salários a processar.

A importação do ficheiro é realizada pela função apresentada na figura 27.

```
void importarFicheiroProcessamento(ListaProcessamentos *lista, char*filename){
    int i = 0;
    lista->numeroSalariosaProcessar=numeroLinhas(filename) + 1;
    FILE *file;
    file = fopen(filename, "r");
    if (lista->numeroSalariosaProcessar > 0) {
    lista->lista_processamentos = (DadosProcessamentoSalario*) malloc(sizeof (DadosProcessamentoSalario) * lista->numeroSalariosaProcessar);
        fscanf(file, "%s %s %s %s %s", lista->desc_codigo_funcionario, lista->desc_numero_dias_trabalhados_semana,
               lista->desc_nr_dias_fim_de_semana, lista->desc_nr_meiodias, lista->desc_faltas);
        while (i < lista->numeroSalariosaProcessar) {
            fscanf(file, "%d,%d,%d,%d,%d", &lista->lista_processamentos[i].codigoFuncionario, &lista->lista_processamentos[i].numero_dias_trabal
                   &lista->lista_processamentos[i].numero_meio_dias_trabalhados, &lista->lista_processamentos[i].numero_faltas);
            i++;
        }
    } else {
        puts("Ficheiro de Importaçao Nulo!");
        lista->numeroSalariosaProcessar = 0;
        lista->lista_processamentos= NULL;
    }
    fclose(file);
}
```

Figura 27 Função que importa um ficheiro de texto com informação para processar salário

Esta função inicia uma *Stream* de leitura, caso o apontador para a *Stream* não seja nulo é obtido o número de linhas do ficheiro, de seguida é alocada a memória necessária para guardar todos dos dados que irão ser utilizados para processar salário. Com a memória alocada, é feito a leitura das linhas e armazenados os dados na estrutura da figura 20. Em caso de a leitura ter falhado, não é alocada memória e o número de salários a processar é colocado a 0.

As estruturas criadas para armazenar os resultados dos processamentos estão apresentadas nas figuras 28 e 29.

```
typedef struct {
    int codigoFuncionario;
    float salarioLiquido;
    float salarioBruto;
    float subsidioAlimentacao;
    float bonus;
    float retencaoIrs;
    float segurancaSocial;
    float impostoSegurancaSocialEmpresa;
    float encargosTotaisEmpresa;

} VencimentoProcessados;
```

Figura 28 Estrutura para guardar um vencimento processado

```
typedef struct{
    char nomeRelatorio[TAMANHODESC];
    int salariosaProcessar;
    int salariosProcessados;
    int salariosNaoProcessados;
    VencimentoProcessados *lista;
}ListaVencimentosProcessados;
```

Figura 29 Estrutura para guarda uma lista de vencimentos processados

A figura 28 guarda todos os valores, em *float,* resultantes do processamento de salário. A figura 29 representa uma lista de vencimentos processados, esta guarda o número de salários processados corretamente e o número de salários não processados (códigos de funcionários que não existem na lista). Armazena também toda a informação resultante dos salários processados, e por fim possui também um *array* de caracteres para guardar o nome do relatório, que é utilizado caso este venha a ser gerado.

```
typedef struct{
    ListaVencimentosProcessadosMes janeiro;
    ListaVencimentosProcessadosMes fevereiro;
    ListaVencimentosProcessadosMes marco;
    ListaVencimentosProcessadosMes abril;
    ListaVencimentosProcessadosMes maio;
    ListaVencimentosProcessadosMes junho;
    ListaVencimentosProcessadosMes julho;
    ListaVencimentosProcessadosMes agosto;
    ListaVencimentosProcessadosMes setembro;
    ListaVencimentosProcessadosMes outubro;
    ListaVencimentosProcessadosMes novembro;
    ListaVencimentosProcessadosMes dezembro;
    int nrMesesProcessados;
}ListaTodosVencimentosProcessados;
```

Figura 30 Estrutura que representa todos os meses

A figura 30 contém 12 estruturas que representam uma lista de vencimentos processados, cada uma representa um mês. Contém também um valor *int* que mostra o número de meses nos quais os salários já foram processados. Quando o utilizador escolhe a opção de processar salário é apresenta uma lista com os 12 meses do ano. Mediante a opção escolhida, os vencimentos processados (caso tenha existido cálculo) são armazenados na estrutura correspondente ao mês escolhido. Por exemplo se for processado o mês de dezembro os vencimentos são guardados na variável “dezembro”.

Funções utilizadas no cálculo do salário

Para a execução dos cálculos de salário foi criado um conjunto funções que calculam determinados valores. O processamento de salário só é possível caso as tabelas e a lista de funcionários se encontrem em memória.

Quando invocada a função que realiza os cálculos, a primeira operação executada é a pesquisa do código do funcionário na lista (função da figura 5). Caso o código não seja encontrado na lista o salário não é processado, sendo guardado essa informação na lista de vencimentos processados (estrutura representada pela figura 29).

Caso o funcionário esteja na lista é devolvido um apontador que será utilizado para obter informações para cálculo (por exemplo valor hora e valor do subsídio de alimentação). A figura 31 representa a função que executa o que foi descrito.

```
void ProcessamentoSalario(ListaProcessamentos listaDados, ListaFuncionarios listaFuncionarios, TabelaIrs tabelaIrs, TabelaSegurancaSocial tabelaSegurancaSocial,
        ListaVencimentosProcessados *listaVencimentosProcessados){
    int i;

    for(i = 0 ; i < listaDados.numeroSalariosaProcessar; i++){
        Funcionario *func = obterFuncionario(&listaFuncionarios,listaDados.lista_processamentos[i].codigoFuncionario);
        if (func != NULL){
            calculoSalario(func,listaDados.lista_processamentos[i],tabelaIrs,tabelaSegurancaSocial,&listaVencimentosProcessados->lista[listaVencimentosProcessados->salariosProcessados]);
            listaVencimentosProcessados->salariosProcessados++;

        }else{
          listaVencimentosProcessados->salariosNaoProcessados++;
        }
    }

}
```

Figura 31 Função que executa cálculo de salário

Entre o conjunto de funções que calculam o salário estão cálculo da retenção de IRS, valor de subsídio de alimentação e o valor a entregar à segurança social.

```
float subsidioAlimentacao(int numeroDiasTrabalhadosSemana, int numeroDiasTrabalhadosFimdesemana, float subsidio) {
    return (numeroDiasTrabalhadosSemana + numeroDiasTrabalhadosFimdesemana) * subsidio;
}
```

Figura 32 Função que calcula subsídio de alimentação

```
float calculoRetencaoIrs(TabelaIrs tabela_irs, Funcionario func, float salario_bruto) {
    int aux = 0, i = 0;
    Irs tabela_pesquisa[tabela_irs.numeroCondicoes];
    //Faz a copia da tabela apenas necessaria
    for (i = 0; i < tabela_irs.numeroCondicoes; i++) {
        if (func.estadoCivil == tabela_irs.listaCriterios[i].ident) {
            tabela_pesquisa[aux] = tabela_irs.listaCriterios[i];
            aux++;
        }
    }
    //Procura nivel salarial
    i = 0;
    while (salario_bruto > tabela_pesquisa[i].nivelSalario && i < aux) {
        i++;
    }
    //Caso o nivel seja o ultimo nivel
    if (i == aux) {
        i--;
    }
    if (func.nrDependentes == 0) {
        return salario_bruto * (tabela_pesquisa[i].semDependentes * VALOR_PER);
    } else if (func.nrDependentes == 1) {
        return salario_bruto * (tabela_pesquisa[i].umDependente * 0.01);
    } else if (func.nrDependentes == 2) {
        return salario_bruto * (tabela_pesquisa[i].doisDependentes * VALOR_PER);
    } else if (func.nrDependentes == 3) {
        return salario_bruto * (tabela_pesquisa[i].tresDependentes * VALOR_PER);
    } else if (func.nrDependentes == 4) {
        return salario_bruto * (tabela_pesquisa[i].quatroDependentes * VALOR_PER);
    } else {
        return salario_bruto * (tabela_pesquisa[i].cincoMaisDependentes * VALOR_PER);
    }

}
```

Figura 33 Função de cálculo de IRS

A função 33, percorre toda a tabela de IRS, mediante a informação sobre o estado civil. Quando encontra a linha correspondente ao nível do salário bruto, verifica o número de dependentes do funcionário e aplica a respetiva taxa.

```
float calculoSegurancaSocial(TabelaSegurancaSocial tabelaSegurancaSocial, Funcionario func, float salarioBruto, float *impostoSegurancaSocialEmpresa) {
    int i = 0, opcao = 0;
    float entidadeEmpregadora, funcionario, global;
    while (i < tabelaSegurancaSocial.numeroLinhas && opcao == 0) {
        if (func.criterioSS == tabelaSegurancaSocial.criterios[i].identicacaoCriterio) {
            opcao = 1;
            entidadeEmpregadora = tabelaSegurancaSocial.criterios[i].EntidadeEmpregadadora *VALOR_PER;
            funcionario = tabelaSegurancaSocial.criterios[i].Trabalhador * VALOR_PER;
            global = funcionario + entidadeEmpregadora;
        }
        i++;
    }
    if (opcao == 0) { //Caso o número de criterio do funcionário não existir na tabela, usa o geral
        entidadeEmpregadora = tabelaSegurancaSocial.criterios[0].EntidadeEmpregadadora *VALOR_PER;
        funcionario = tabelaSegurancaSocial.criterios[0].Trabalhador * VALOR_PER;
        global = funcionario + entidadeEmpregadora;
    }
    *impostoSegurancaSocialEmpresa = salarioBruto * entidadeEmpregadora; //Guarda o Valor que a empresa tem de pagar
    return salarioBruto * global;
}
```

Figura 34 Função que calcula a contribuição de Segurança Social

Esta função, através do número da taxa presente no funcionário, pesquisa na tabela os valores da taxa a aplicar. Caso o funcionário tenha uma identificação de taxa que não existe na tabela é usada a taxa geral.

Exemplo: Funcionário com Idtaxa 3.

```
IdTaxa TrabalhadoresPorContaOutrem EEmpregadora Trabalhador GLobal
1,Trabalhadores_em_geral 23.75,20.00,43.75
2,Membros_dos_órgãos_estatutários_das_pessoas_coletivas_em_geral 23.50,3.10,26.60
3,Membros_dos_órgãos_estatutários_que_exerçam_funções_de_gerência_ou_de_administração 23.75,5.00,28.75
```

A função compara o valor presente no funcionário com os valores das linhas. Quando o valor é igual aplica as taxas.

Caso o funcionário tenha uma taxa com o número 4 (número que não existe na tabela) é aplicada a taxa 1, que é a taxa geral. Caso o utilizador inserira a taxa 4 na tabela, esta passa a ser aplicada.

Bónus por idade, antiguidade e assiduidade

No cálculo do salário foi também adicionado o bónus. O bónus tem em conta três dados, a idade, o tempo na empresa e assiduidade. Para a obtenção do bónus foi criada uma função que calcula idades. Esta função obtém o dia atual através do computador e realiza o cálculo quer com a data de nascimento quer com a data de entrada na empresa.

```
int calcularIdade(Data data) {
    time_t t = time(NULL);
    struct tm *tm = localtime(&t);
    int idade;

    if (tm->tm_mon + 1 < data.mes) {
        idade = tm->tm_year + 1900 - (data.ano + 1);
    } else if (tm->tm_mon + 1 > data.mes) {
        idade = tm->tm_year + 1900 - data.ano;
    } else {
        if (data.dia < tm->tm_mday) {
            idade = tm->tm_year + 1900 - data.ano;
        } else if (data.dia > tm->tm_mday) {
            idade = tm->tm_year + 1900 - (data.ano + 1);
        } else {
            idade = tm->tm_year + 1900 - data.ano;
        }
    }
    return idade;
}
```

Figura 35 Função que calcula idade

Os bónus são atribuídos em percentagem de acordo com a idade e tempo na empresa. Foi tido em conta também o número de faltas do funcionário, caso tenha a folha limpa é recompensado. Em caso de faltas é penalizado, sendo que o bónus pode mesmo ser negativo. Nesse caso em vez de ser adicionado um valor ao salário bruto, é subtraído.

```
    //Assuidade
    if (numeroFaltas == 0) { //Nunca falta + bonus de 0.5%
        bonus = bonus + 0.5;
    } else if (numeroFaltas == 1) {// uma falta - 2%
        bonus = bonus - 2;
    } else if (numeroFaltas == 2 || numeroFaltas == 3) { // duas ou 3 faltaS - 2%
        bonus = bonus - 4.5;
    } else if (numeroFaltas > 3) { // 3 ou mais faltas
        bonus = bonus - (numeroFaltas * 1.5);
    }
    return salarioBruto * (bonus * VALOR_PER);
}
```

Figura 36 Bónus por assiduidade

O valor a receber pelo funcionário nos dias trabalhados ao fim de semana é também aumentado em 50%.

Escrita dos vencimentos processados em ficheiro

A escrita do relatório em ficheiro é feita caso o utilizador assim o pretenda. Para a escrita ser realizada é necessário que tenha ocorrido um processamento de salário. No caso de já existir um relatório referente ao mês processado, é substituído. Quando escrito em ficheiro, o relatório pode importado.

2.5 Memória Dinâmica

A memória dinâmica foi utilizada sempre que as funcionalidades assim o requisitavam. Todas as funções de adicionar um novo elemento a uma lista verificam o tamanho da memória existente e o número de dados armazenados nessa memória. No caso de o limite ser atingido, é feita uma realocação duplicando o tamanho existente. A figura 30 mostra a função que executa uma realocação, neste para a lista de funcionários.

```
void expandirMemoria(ListaFuncionarios *listafuncionarios){
    Funcionario *func = (Funcionario*) realloc(listafuncionarios->listafuncionarios, sizeof(Funcionario) * (listafuncionarios->tamanho * 2));
    listafuncionarios->listafuncionarios= func;
    listafuncionarios->tamanho *= 2;

}
```

Figura 37 Expansão de memória lista de funcionários

Foi também implementado as funções que libertam a memória alocada, estas funções são utilizadas em duas situações. A primeira quando o utilizador fecha o programa, a segunda quando é feita uma leitura de ficheiro, na qual toda a memória do computador é substituída pela informação em ficheiro. A figura 38 mostra as funções que são invocadas quando o utilizador lê ficheiro. Nesta figura é possível visualizar que são invocadas as funções que libertam a memória alocada. Quando a leitura de ficheiro é feita é alocada memória de acordo com os dados em ficheiro

```
case  4:
    //Liberta a memória alocada, caso alocada e lê os ficheiros
    if(listaFuncionarios->tamanho != 0){libertarMemoriaFuncionarios(listaFuncionarios);}
    if(tabelaIrs->tamanho != 0){  libertarMemoriaTabelaIRS(tabelaIrs);}
    if(tabelaSegurancaSocial->tamanho != 0){libertarMemoriaTabelaSS(tabelaSegurancaSocial);}
    if(relatorioFaltas.estadoRelatorio == 1){libertarMemoriaGestaoFaltas(&relatorioFaltas);}
    libertarMemoriaRelatoriosProcessados(relatoriosVencimentosProcessados);
    obterRelatorioDespesasTotais(&despesas,FILENAME_DESPESASTOTAIS);
    obterRelatoriopercentagemIdadesFuncionarios(&relatorioIdade,FILENAME_PERCENTAGEMIDADES);
    obterRelatorioPercentagemTempoEmpresa(&relatorioTempoEmpresa,FILENAME_PERCENTAGEMANOSEMPRESA);
    obterRelatorioFaltas(&relatorioFaltas,FILENAME_RELATORIOFALTAS);
    logMsg("Leitura Relatórios Processados", LOG_FILE);
    obterTodosRelatoriosProcessados(relatoriosVencimentosProcessados);
    logMsg("Leitura Ficheiro Segurança Social", LOG_FILE);
    lerFicheiroSegurancaSocial(tabelaSegurancaSocial, FILE_SEGURANCA_SOCIAL);
    logMsg("Leitura Ficheiro Funcionários", LOG_FILE);
    carregarFuncionariosFx(listaFuncionarios, FILE_FUNCIONARIOS);
    logMsg("Leitura Ficheiro IRS", LOG_FILE);
    lerFicheiroIrs(tabelaIrs, FILEIRS);
    if (verificaExistenciaFicheiro(FILENAME_RELATORIOVALORH) == 1) {
        obterRelatorioValorHora(&relatorioValorhora, FILENAME_RELATORIOVALORH);
    }
    if (verificaExistenciaFicheiro(FILENAME_RELATORIOESTADOCIVIL) == 1) {
        obterRelatorioPercentagemEstadoCivil(&relatorioEstadoCivil, FILENAME_RELATORIOESTADOCIVIL);
    }
```

Figura 38 Função invocada quando o utilizado lê ficheiros

2.6 Persistência de dados

A persistência de dados foi referida aquando a descrição das funcionalidades, e consiste na escrita/leitura das informações nos ficheiros. Para esta funcionalidade foi definido que não é possível ao utilizador gravar uma lista sem funcionários, ou seja, para que a lista seja guardada tem de ter pelos menos um funcionário criado. Foi definido ainda que um relatório nulo não é guardado.

Enumerações

Na criação das estruturas de dados foram criadas três enumerações. A decisão de implementar as enumerações está relacionada com o facto de o funcionário possuir dados que correspondem a valores predefinidos. Por exemplo, o funcionário só poderá ter dois estados, ativo ou removido.

```
typedef enum {
    NAOCASADO = 1,
    CASADOUNICOTITULAR,
   CASADODOISTITULARES
} EstadoCivil;

typedef enum {
    ADMINISTRADOR = 1, EMPREGADO, CHEFE
} Cargo;

typedef enum {
     REMOVIDO = 0,ATIVO
} EstadoFuncionario;
```

Figura 39 Enumerações

Para além da enumeração do estado do funcionário, foi criada uma enumeração para o estado civil e outra para o cargo.

3. Funcionalidades propostas

3.1 Percentagem de funcionários por intervalo de idades

A listagem de percentagens de funcionários por intervalos de idades foi implementada porque o grupo considerou que era uma informação relevante nomeadamente para obter o número de funcionários que recebem bónus pela idade. Com esta listagem é também possível saber a percentagem de funcionários acima dos 60 anos, que por norma é uma idade em que já se pode obter a pré-reforma.

Para implementação desta listagem foi criada uma estrutura, figura 40.

```
typedef struct {
    int funcionariosAnalisados;
    float percentagemMenos26;
    float percentagemEntre26e30;
    float percentagemEntre31e40;
    float percentagemEntre41e50;
    float percentagemEntre51e60;
    float percentagemMais60;
    int funcionariosComBonus;
    int funcionariosSemBonus;
} PercentagemIdades;
```

Figura 40 Estrutura para armazenar as percentagens de funcionários por intervalo de idades

Esta estrutura possui quatro dados do tipo *float*, para guardar as percentagens. Possui três dados do tipo *int* que guardam o número de funcionários analisados, o número de funcionários que obtém bónus por idade, e o número dos que não obtém.

Para que o utilizador consiga obter esta listagem é necessário que exista uma lista de funcionários, caso contrário a invocação da função que gera a listagem não é realizada.

Função implementada

```
void percentagemIdadesFuncionarios(ListaFuncionarios lista, PercentagemIdades *percentagens) {
    int maissessenta = 0, entre5le60 = 0, entre4le50 = 0, entre3le40 = 0, entre26e30 = 0, menos26 = 0;
    int idade;
    for (int i = 0; i < lista.nrFuncionarios; i++) {
        idade = calcularIdade(lista.listafuncionarios[i].dataNascimento);
        printf("%d\n\n", idade);
        if (idade > 60) { //Trabalhador com + de 60 anos
            maissessenta++;
        } else if (idade <= 60 && idade > 50) {//Trabalhador entre 51 e 60 anos
            entre5le60++;
        } else if (idade <= 50 && idade > 41) { //Trabalhador entre 41 e 50 anos
            entre4le50++;
        } else if (idade <= 40 && idade > 30) { //Trabalhador entre 31 e 40 anos
            entre3le40++;
        } else if (idade <= 30 && idade > 25) { // Trabalhador entre 26 e 30 anos
            entre26e30++;
        } else if (idade < 26) {
            menos26++;
        }
    }
    percentagens->funcionariosAnalisados = lista.nrFuncionarios;
    percentagens->percentagemMais60 = (float) maissessenta / lista.nrFuncionarios;
    percentagens->percentagemMais60 *= 100;
    percentagens->percentagemEntre5le60 = (float) entre5le60 / lista.nrFuncionarios;
    percentagens->percentagemEntre5le60 *= 100;
    percentagens->percentagemEntre4le50 = (float) entre4le50 / lista.nrFuncionarios;
    percentagens->percentagemEntre4le50 *= 100;
    percentagens->percentagemEntre3le40 = (float) entre3le40 / lista.nrFuncionarios;
    percentagens->percentagemEntre3le40 *= 100;
    percentagens->percentagemEntre26e30 = (float) entre26e30 / lista.nrFuncionarios;
    percentagens->percentagemEntre26e30 *= 100;
    percentagens->percentagemMenos26 = (float) menos26 / lista.nrFuncionarios;
    percentagens->percentagemMenos26 *= 100;
    percentagens->funcionariosComBonus = entre26e30 + entre3le40 + entre4le50 + entre5le60 + maissessenta;
    percentagens->funcionariosSemBonus = menos26;

}
```

Figura 41 Função que obtém percentagens de funcionários por idades

Esta função percorre toda a lista de funcionários, guardando o número de funcionários pelo intervalo de idade. De seguida é calculada a percentagem que cada intervalo de idades representa.

O intervalo de idades foi definido pelo bónus a receber por idade, ou seja, o intervalo de idades corresponde a um valor do bónus.

Exemplo: Idade menor que 26 anos -> não tem bónus;

Idade entre 26 e 30 anos-> bónus de 5.5%;

3.2 Percentagem de funcionários por intervalo de anos na empresa

A listagem de percentagens de funcionários por anos na empresa foi implementada porque o grupo considerou que era uma informação relevante nomeadamente para obter o número de funcionários que recebem bónus por tempo na empresa e os que não recebem.

Para implementação desta listagem foi criada uma estrutura, figura 42.

```
typedef struct {
    int funcionariosAnalisados;
    float percentagemMenos5;
    float percentagemEntre5e10;
    float percentagemEntre11e20;
    float percentagemMais20;
    int funcionariosComBonus;
    int funcionariosSemBonus;
} PercentagemTempoEmpresa;
```

Figura 42 Estrutura para armazenar as percentagens de funcionários por intervalo de anos na empresa

Esta estrutura possui quatro dados do tipo *float*, para guardar as percentagens. Possui três dados do tipo *int* que guardam o número de funcionários analisados, o número de funcionários que obtém bónus por tempo na empresa, e o número dos que não obtém.

Tal como na listagem de percentagens por idade, caso não exista uma lista que possua funcionários, é impossível obter este relatório.

Função implementada

A função implementada para esta listagem é semelhante à percentagem por idades, ou seja, é percorrida a lista de funcionários e obtidos os números de funcionários por intervalo de anos na empresa. Depois é calculada a percentagem.

3.3 Despesas totais da empresa

A listagem de despesas totais da empresa foi implementada, pois é uma informação, que o grupo considerou, bastante importante. Com esta listagem a empresa consegue saber os custos totais com os salários e impostos que teve por mês, bem como o total de todos os meses processados. Esta listagem é dependente do relatório de vencimentos processados.

Para implementação desta listagem foram criadas duas estruturas, figura 43e 44.

```
typedef struct {
    int salariosProcessados;
    float despesaTotal;
} DespesaMes;
```

Figura 43 Estrutura que guarda despesa total de um mês

Esta estrutura possui o número de salários que foram processados do tipo *int* e o valor total que estes encarregam para a empresa.

```
typedef struct {
    DespesaMes meses[12]; //Caso estático, serão sempre analisados os 12 meses
    float despesaTotal;
    RelatorioGerado estadoRelatorio;
} DespesasAnuais;
```

Figura 44 Estrutura utilizada para guardar as despesas por mês e a despesa total

A estrutura "Despesas Anuais" guarda o valor da despesa de cada mês, daí possuir um array com 12 posições. Neste caso não foi utilizada memória dinâmica porque, mesmo que não exista relatório sobre o mês, é gerada informação (caso não haja relatório de vencimentos processados, é escrito nas despesas anuais que o mês ainda não foi processado). Esta estrutura possui também um valor do tipo *float* que é utilizado para guardar a despesa total. Por fim existe uma enumeração. Esta enumeração é utilizada para verificar se durante a execução do programa já foi criado algum relatório.

Função implementada

Para a implementação listagem, os relatórios processados que foram guardados em ficheiro são importados para o programa. Para a importação ser possível foi criada uma função que lê os relatórios e armazena os valores numa variável, com a estrutura da figura 23. Depois de importados os relatórios é calculado a despesa total que a empresa teve com todos os salários e armazenado o número de salários processados. O custo total por mês é também adicionado à variável que guarda o valor de todos os meses.

Caso exista um mês que não tenha relatório criado, é atribuído 0 como despesa.

```
ListaVencimentosProcessados lista;
despesas->despesaTotal = 0;
for (int i = 0; i < 12; i++) {
    if (i == 0) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_JAN);
    } else if (i == 1) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_FEV);
    } else if (i == 2) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_MAR);
    } else if (i == 3) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_ABR);
    } else if (i == 4) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_MAI);
    } else if (i == 5) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_JUN);
    } else if (i == 6) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_JUL);
    } else if (i == 7) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_AGO);
    } else if (i == 8) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_SET);
    } else if (i == 9) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_OUT);
    } else if (i == 10) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_NOV);
    } else if (i == 11) {
        strcpy(lista.nomeRelatorio, NOME_RELATORIO_DEZ);
    }
    obterRelatorio(&lista);
    if (lista.salariosProcessados == 0) {
        despesas->meses[i].despesaTotal = 0;
        despesas->meses[i].salariosProcessados = 0;
    } else {
        despesaTotalEmpresaPorMes(lista, &despesas->meses[i].despesaTotal, &despesas->meses[i].salariosProcessados);
        despesas->despesaTotal += despesas->meses[i].despesaTotal;
    }
}
despesas->estadoRelatorio = 1;
```

Figura 45 Função que calcula custo da empresa por mês

3.4 Listagem sobre o valor hora

Foi considerado também que uma listagem que obtém o valor mínimo que um funcionário recebe por hora, o máximo e o valor médio são do interesse da empresa. Com estes dados a empresa consegue saber o valor máximo e mínimo que paga por hora, e também a diferença de remuneração entre o funcionário que ganha o máximo e o que ganha o mínimo.

Para esta listagem foi criada uma estrutura, figura 46.

```
typedef struct {
    float valorMaximoHora;
    float valorMinimoHora;
    float valorMedio;
    float diferenca;
    RelatorioGerado estadoRelatorio;
} RelatorioValorHora;
```

Figura 46 Estrutura criada para o relatório do valor/hora

Função implementada

A função implementada para obter o valor máximo e mínimo por hora percorre a lista de funcionários e compara os valores. No decorrer da iteração são somados todos os valores hora. Quando a iteração termina, o resultado da soma dos valores é divida pelo número de funcionários, obtendo-se assim o valor médio pago por hora.

Ainda nesta função é calculada a diferença entre o funcionário que ganha o mais e o que ganha o menos.

3.5 Percentagem por estado civil

Foi implementada uma listagem que apresenta as percentagens de funcionários por estado civil. O estado civil influência as contribuições fiscais, essa foi a principal razão pelo qual esta funcionalidade foi inserida.

Para esta listagem foi criada uma estrutura, figura 47.

```
typedef struct {
    float percNaoCasado;
    float percCasadoUnicoTitular;
    float percCasadoDoisTitulares;
    RelatorioGerado estadoRelatorio;
} PercentagemEstadoCivil;
```

Figura 47 Estrutura criada para listagem do estado civil

Função implementada

A função implementada para esta listagem percorre a lista de funcionários e obtém o número de funcionários por estado civil. Depois é calculada a percentagem correspondente a cada estado civil.

3.6 Listagem de funcionários pelo seu número de faltas por mês

A listagem de faltas apresenta, por mês, os funcionários que faltaram uma vez, duas ou três vezes, mais de três e os que não faltaram. Esta informação foi considerada pelo grupo como sendo do interesse da empresa pois permite, por exemplo, analisar os funcionários que mais faltam em meses de trabalho importantes para a empresa. Esta listagem está dependente dos ficheiros que são importados para processar salário.

Foram criadas duas estruturas de dados, figuras 48 e 49.

```
typedef struct {
    int tamanhoSemfaltas;
    int tamanhoUmafaltas;
    int tamanhoDuasouTresFaltas;
    int tamanhoMaisTresFaltas;
    int nrSemfaltas;
    int nrUmafaltas;
    int nrMaisTresFaltas;
    int nrDuasouTresFaltas;
    int *Semfaltas;
    int *Umafaltas;
    int *DuasouTresFaltas;
    int *MaisTresFaltas;
    int nrFuncionariosAnalisados;
} RelatorioFaltasMes;
```

Figura 48 Estrutura que armazena as faltas dos funcionários no mês

A estrutura contém 4 apontadores para valores inteiros, que serão utilizados para criar listas para armazenar os códigos dos funcionários. Contém também o tamanho de cada apontador (que irá ser utilizado para alocar memória). Por fim contém também número de funcionários que foram armazenados em cada lista.

```
typedef struct {
    gestaoFaltasMes lista[12]; //Caso está
    RelatorioGerado estadoRelatorio;
} RelatorioFaltas;
```

Figura 49 Estrutura que armazena o relatório de faltas de todos os meses

A estrutura da figura 49 contém um *array* que armazena o relatório de faltas de cada mês. Contém também a enumeração que permite saber se durante a execução do programa já existe em memória uma listagem sobre as faltas.

Função implementada

Para implementar esta listagem foi utilizada a função de importar ficheiros para processamento, que já tinha sido implementada. Esta função é executada 12 vezes (uma por cada mês). Caso o mês não tenha um ficheiro com dados para processamento, o relatório por faltas não é executado. Caso tenha, são analisadas todas as linhas, e de seguida, o código do funcionário é colocado na lista correspondente (mediante o número de faltas que teve). Como as listas que guardam os códigos são implementadas com memória dinâmica, caso haja necessidade de aumentar o tamanho da memória, é feita a realocação.

```
//Verifica se ficheiro existe
if (verificaExistenciaFicheiro(FicheiroImport) == 1) {
    importarFicheiroProcessamento(lista, FicheiroImport);
    gestaofaltas->lista[i].Umafaltas = (int*) malloc(sizeof (int) * TAMANHOCODIGOS);
    gestaofaltas->lista[i].Semfaltas = (int*) malloc(sizeof (int) * TAMANHOCODIGOS);
    gestaofaltas->lista[i].DuasouTresFaltas = (int*) malloc(sizeof (int) * TAMANHOCODIGOS);
    gestaofaltas->lista[i].MaisTresFaltas = (int*) malloc(sizeof (int) * TAMANHOCODIGOS);
    gestaofaltas->lista[i].nrFuncionariosAnalisados = lista->numeroSalariosaProcessar;
    for (int j = 0; j < lista->numeroSalariosaProcessar; j++) {
        if (lista->lista_processamentos[j].numero_faltas == 0) {
            if (gestaofaltas->lista[i].nrSemfaltas == gestaofaltas->lista[i].tamanhoSemfaltas) {
                expandirMemoriaFaltas(gestaofaltas->lista[i].Semfaltas, &gestaofaltas->lista[i].tamanhoSemfaltas);
            }
            gestaofaltas->lista[i].Semfaltas[gestaofaltas->lista[i].nrSemfaltas] = lista->lista_processamentos[j].codigoFuncionario;
            gestaofaltas->lista[i].nrSemfaltas++;
        } else if (lista->lista_processamentos[j].numero_faltas == 1) {
            if (gestaofaltas->lista[i].nrUmafaltas == gestaofaltas->lista[i].tamanhoUmafaltas) {
                expandirMemoriaFaltas(gestaofaltas->lista[i].Umafaltas, &gestaofaltas->lista[i].tamanhoUmafaltas);
            }
            gestaofaltas->lista[i].Umafaltas[gestaofaltas->lista[i].nrUmafaltas] = lista->lista_processamentos[j].codigoFuncionario;
            gestaofaltas->lista[i].nrUmafaltas++;
        } else if (lista->lista_processamentos[j].numero_faltas == 2 || lista->lista_processamentos[j].numero_faltas == 3) {
            if (gestaofaltas->lista[i].nrDuasouTresFaltas == gestaofaltas->lista[i].tamanhoDuasouTresFaltas) {
                expandirMemoriaFaltas(gestaofaltas->lista[i].DuasouTresFaltas, &gestaofaltas->lista[i].tamanhoDuasouTresFaltas);
            }
            gestaofaltas->lista[i].DuasouTresFaltas[gestaofaltas->lista[i].nrDuasouTresFaltas] = lista->lista_processamentos[j].codigoFuncionario;
            gestaofaltas->lista[i].nrDuasouTresFaltas++;
        } else if (lista->lista_processamentos[j].numero_faltas > 3) {
            if (gestaofaltas->lista[i].nrMaisTresFaltas == gestaofaltas->lista[i].tamanhoMaisTresFaltas) {
                expandirMemoriaFaltas(gestaofaltas->lista[i].DuasouTresFaltas, &gestaofaltas->lista[i].tamanhoMaisTresFaltas);
            }
            gestaofaltas->lista[i].MaisTresFaltas[gestaofaltas->lista[i].nrMaisTresFaltas] = lista->lista_processamentos[j].codigoFuncionario;
            gestaofaltas->lista[i].nrMaisTresFaltas++;
        }
    }
```

Figura 50 Função que lista os funcionários pelo seu número de faltas

Escrita em Ficheiro

Todas as listagens têm uma função que permitem gerar um relatório em ficheiro. O relatório é gerado apenas se as listagens forem executadas durante programa. A figura 51 mostra a função que guarda em ficheiro a percentagem dos funcionários por idades.

```
void gerarRelatoriopercentagemIdadesFuncionarios(PercentagemIdades *percentagens, char *filename) {
    FILE *file;
    file = fopen(filename, "w");
    char percentagem [2] = "%";
    if (file != NULL) {
        fprintf(file, "Número de Funcionarios analisados: %d \n", percentagens->funcionariosAnalisados);
        fprintf(file, "Número de Funcionarios com Direito a Bónus por idade: %d \n", percentagens->funcionariosComBonus)
        fprintf(file, "Número de Funcionarios sem Bónus por idade: %d \n", percentagens->funcionariosSemBonus);
        fprintf(file, "Mais de 60 anos: %.2f%s\n", percentagens->percentagemMais60, percentagem);
        fprintf(file, "Entre 51 e 60 anos: %.2f%s\n", percentagens->percentagemEntre51e60, percentagem);
        fprintf(file, "Entre 41 e 50 anos: %.2f%s\n", percentagens->percentagemEntre41e50, percentagem);
        fprintf(file, "Entre 26 e 30 anos: %.2f%s\n", percentagens->percentagemEntre31e40, percentagem);
        fprintf(file, "Entre 31 e 40 anos: %.2f%s\n", percentagens->percentagemEntre26e30, percentagem);
        fprintf(file, "Menos de 26 anos: %.2f%s\n", percentagens->percentagemMenos26, percentagem);
    } else {
        puts("Impossivel Gerar relatório da percentagem de Idades!");
    }

    fclose(file);
}
```

Figura 51 Função que gera relatório de listagem em ficheiro

Leitura de Ficheiro

Todos os relatórios podem também ser importados do ficheiro para a memória do computador.

O utilizador pode visualizar todos os relatórios em memória e caso pretenda nova listagem pode também substitui-los.

4. Estrutura analítica do projeto

O planeamento do projeto foi realizado tem em conta dois fatores, nível de conhecimento e requisitos de cada funcionalidade.

O nível de conhecimento exigido para a realização do projeto obrigou a que as tarefas fossem definidas consoante o conhecimento que era obtido nas aulas de Fundamentos de Programação.

Os requisitos que cada funcionalidade apresentava foram também tidos em conta no planeamento, por exemplo, para a funcionalidade de processamento de salário é necessário já ter as tabelas de Segurança social e IRS importadas, bem como uma lista de funcionários.

Com isto foi definido as seguintes tarefas.

1->Tarefa: Gestão de Funcionários

Sub-tarefas:

1) Criação da estrutura de dados;
2) Criação de uma estrutura (lista estática) para armazenamento dos funcionários;
3) Criação dos inputs de inserção de dados sobre um funcionário (leitura datas, leitura do número de telemóvel, estado civil, cargo, etc.);
4) Implementação da função de adição de um funcionário à lista;
5) Implementação da função de remoção de um funcionário da lista;
6) Implementação da função de *update* de um funcionário da lista;
7) Listagem de dados de um funcionário pelo seu código;
8) Escrita da lista de funcionários em ficheiro;
9) Leitura da lista de funcionários de ficheiro;

As sub-tarefas apresentadas mostram a ordem pelas quais foram implementadas, sendo que a sub-tarefa 4 dependiam da 1, 2 e 3. As sub-tarefas 5 6 e 7 dependem da 4.

2->Tarefa: Gestão de Tabela de IRS

Sub-tarefas:

1) Criação das estruturas de dados;
2) Importação da tabela do ficheiro para o programa
3) Implementação da função de adição de uma linha à tabela;
4) Implementação da função de remoção de uma linha da tabela;
5) Implementação da função de *update* de uma linha da tabela;
6) Escrita da tabela em ficheiro.

As sub-tarefas apresentadas mostram a ordem pelas quais foram implementadas, sendo que as sub-tarefas 3,4,5, 6 dependiam da 1 e 2.

3->Tarefa: Gestão de Tabela de Segurança Social.

Sub-tarefas:

1) Criação das estruturas de dados;
2) Importação da tabela do ficheiro para o programa
3) Implementação da função de adição de uma linha à tabela;
4) Implementação da função de remoção de uma linha da tabela;
5) Implementação da função de *update* de uma linha da tabela;
6) Escrita da tabela em ficheiro.

As sub-tarefas apresentadas mostram a ordem pelas quais foram implementadas, sendo que as sub-tarefas 3,4,5, 6 dependiam da 1 e 2.

4->Tarefa: Processamento de Salário

Sub-tarefas:

1) Criação das estruturas de dados solicitadas;
2) Importação do ficheiro de salários a processar;
3) Criação das funções de cálculo de salário;
4) Criação das funções que executam o cálculo;
5) Armazenamento dos salários processados;
6) Criação do relatório em ficheiro.

As sub-tarefas apresentadas mostram a ordem pelas quais foram implementadas, sendo que a sub-tarefa 2 dependia da 1, a 4 da 3, a 5 da 4, e a 6 da 5. A tarefa processamento de salário é dependente das três funcionalidades em cima descritas.

5->Tarefa: Inclusão da memória dinâmica

Sub-tarefas:

1) Criação das funções de alocação, expansão e libertação de memória;
2) Implementação da memória dinâmica na lista de funcionários;
3) Implementação da memória dinâmica na tabela de IRS;
4) Implementação da memória dinâmica na tabela de Segurança Social;
5) Implementação da memória dinâmica na importação dos dados para processamento de salário;
6) Implementação da memória dinâmica na lista de vencimentos processados;

As sub-tarefas 2,3,4,5,6 dependia da realização da 1.

6->Tarefa: Listagens propostas pelo grupo

Sub-tarefas:

1) Listagem de funcionários pelo seu número de faltas por mês;
2) Percentagem por estado civil;
3) Listagem sobre o valor hora;
4) Despesas totais da empresa;
5) Percentagem de funcionários por intervalo de idades;
6) Percentagem de funcionários por tempo na empresa;
7) Implementação das funções que geram relatório em ficheiro;
8) Implementação das funções que importam os relatórios de ficheiro;

Esta tarefa dependia das tarefas de gestão de funcionários e de processamento de salário.

A tarefas de gestão de funcionários e as listagens propostas pelo grupo foram realizadas pelo aluno Diogo.

O processamento de salário, importação/gestão das tabelas, e a criação dos relatórios de vencimentos foram da responsabilidade do outro elemento do grupo.

Planeamento Temporal	Inicio	Fim
Gestão de funcionários	11/11/2020	27/11/2020
Gestão de Tabela de IRS	23/11/2020	12/12/2020
Gestão de Tabela de SS	23/11/2020	16/10/2020
Processamento de Salário	18/12/2020	3/1/2021
Listagens	3/1/2021	12/1/2021

5. Funcionalidades implementadas

As funcionalidades implementadas correspondem com as funcionalidades inicialmente propostas. Fazendo uma execução do programa é possível verificar isso.

Tarefas

Gestão de funcionários-> Todas as funcionalidades propostas para a gestão de funcionários foram implementadas.

Gestão da tabelas -> É possível importar e alterar as tabelas de IRS e SS, como proposto.

Processamento de salário-> O programa é capaz de importar um ficheiro, efetuar os cálculos de salário e gerar relatório de vencimentos.

Persistência de dados-> Todos os relatórios gerados podem ser gravados em ficheiro, tal como as tabelas e a lista de funcionários.

Memoria Dinâmica-> Sempre que as funcionalidades requeriam, foi implementada memória dinâmica.

Listagens de dados-> O grupo desenvolveu 6 listagens de dados que foram descritos no ponto 3 deste documento, todas as listagens podem ser guardadas/importadas de ficheiro.

Para conclusão, todas as funcionalidades propostas foram implementadas no projeto.

6. Conclusão

A realização deste projeto permitiu ao grupo desenvolver conhecimentos na linguagem C. Ao longo da realização do projeto foram sentidas algumas dificuldades que foram ultrapassadas com a pesquisa de informação e documentação.

O planeamento foi bastante importante, visto que permitiu que a elaboração do projeto decorresse de uma forma orientada e organizada.

Página utilizada para testes

Nos testes dos cálculos do processamento de salário foi usada a página “doutor finanças”.

https://www.doutorfinancas.pt/simulador-salario-liquido-2020/