**INTRODUÇAO**

Seguindo os exercícios das aulas do semestre sobre o  **“management”** de funcionários de uma empresa e respetivos salários, este projeto pretende criar um programa mais completo e mais adequado ao uso profissional. Assim, é necessário uma base de dados sobre funcionários mais completa e dinâmica, uma facilitação de uso do sistema por qualquer indivíduo permitindo implementar informação através de ficheiros de texto, e um conjunto de informações geradas pelo sistema mais completo e prático.

Este grupo optou por existirem 6 funcionalidades principais apresentadas no menu. Cálculo de salários dos funcionários no sistema, gestão desses mesmos funcionários e gestão das tabelas relativas aos descontos do IRS e às taxas de segurança social necessários paras os cálculos. Para além dessas opções, também permite ter acesso às 5 listagens que o grupo considerou pertinentes de existirem, e contém uma opção de salvar ou eliminar toda a informação até aí acrescentada/modificada na memória do programa.

O grupo decidiu que os dados sobre os funcionários e os necessários a cálculos salariais podiam ser inseridos tanto manualmente por inputs como por ficheiros de texto incluídos na pasta UTILIZADOR do programa. Ao mesmo tempo, que haveriam então três arrays de memoria principais a serem usados, o dos funcionários **arrayRH.funcionarios\_array**, o dos dados para cálculos salariais **conta.calculo\_array**, e o dos processamentos de salários realizados **salarios.calculo\_array**, e comum a estes três arrays de memoria seria que as suas structs respetivas teriam a variável código, sendo esta o código do funcionário e a maneira de encontrar os respetivos dados e processamentos. Estes arrays de memoria seriam iniciados com um espaço logo no início, e depois acrescentados um novo espaço depois desse ser ocupado com dados guardados.

Em relação ao desempenho do programa, decidiu-se que logo no inicio de cada utilização seriam impressos o numero de espaços ocupados pelos três arrays de memoria principais no programa e se as tabelas correspondentes a IRS e Segurança Social tinham sido importadas com sucesso, importando os ficheiros users.bin, salarios.bin, SALARIOS\_PROCESSADOS\_bin, DEPENDENTE\_CASADO\_DOIS\_TITULARES\_bin, DEPENDENTE\_CASADO\_UNICO\_TITULAR\_bin, DEPENDENTE\_NAO\_CASADO\_bin e TAXAS.txt para a memoria do computador, pronta a ser usada. No final do programa, sempre que fosse escolhida a opção de sair, seria perguntado ao utilizador se não deseja guardar a informação toda de volta para os ficheiros do programa.

**(faltam decisões tomadas)**

**FUNCIONALIDADES REQUERIDAS**

**ESTRUTUTRA ANALITICA DO PROJETO**

**FUNCIONALIDADES IMPLEMENTADAS**

**CONCLUSAO**

Para o armazenamento de funcionários na base de dados, os parâmetros sobre cada funcionário foram incluídos numa struct **Funcionario**, como **ativo** que indica se o funcionário está ativo no sistema (1) ou não pois foi removido pelo utilizador (0). Esta por sua vez inclui três parâmetros do tipo **Data**, uma struct com variáveis para datas, e duas enumerações, uma do tipo **Est\_Civil** e outra do tipo **Cargo**. Uma outra struct foi criada, neste caso tipo **Empresa**, que inclui um contador para manter conta do número de funcionários, e o pointer de um array de memória do tipo da struct Funcionario, **funcionarios\_array**. Este array é inicializado com um espaço de memoria da struct Funcionario quando no main é criado uma variável tipo Empresa chamada **RH**, cujo contador começa em 0. É o pointer de RH que é logo passado para a função menu.

Para o processamento de cálculos salariais na base de dados, os parâmetros necessários para cada cálculo, como **estado,** que indica se já foi processado (1) ou não (0), ou se a struct é invalida por alguma razão (-1). foram incluídos numa struct **Conta**. Uma outra struct foi criada, neste caso tipo **Lista\_calc**, que inclui um contador para manter conta do número de cálculos, e o pointer de um array de memória do tipo da struct Conta, **calculo\_array**. Este array é inicializado com um espaço de memoria da struct Conta quando no main é criado uma variável tipo Lista\_calc chamada **conta**, cujo contador começa em 0. É o pointer de conta que é logo passado para a função menu.

Para o processamento de cálculos salariais na base de dados, os parâmetros envolvidos em cada processamento salarial foram incluídos numa struct **Calculo**. Uma outra struct foi criada, neste caso tipo **Lista\_salarios**, que inclui um contador para manter conta do número de processamentos, e o pointer de um array de memória do tipo da struct Calculo, **calculo\_array**. Este array é inicializado com um espaço de memoria da struct Calculo quando no main é criado uma variável tipo Lista\_salarios chamada **salarios**, cujo contador começa em 0. É o pointer de salarios que é logo passado para a função menu.

Para o registo das taxas de IRS, foram criadas três tabelas de valores, uma para os de dois titulares, outra para os de um titular e outra para os não casados. Os parâmetros em cada linha dessas tabelas incluem o vencimento do funcionario e as várias opções do numero de filhos, que se encontram dentro de structs, **Dois\_titulares**, **Unico\_titular** e **Nao\_casado**, respetivamente. Outras tres structs foram criadas, do tipo **ListaUm**, **ListaDois** e **ListaTres**, respetivamente, que incluem um contador para manter conta do numero de linhas em cada tabela, e o pointer de um array de memoria do tipo da struct respetiva. Estes arrays são inicializados com um espaço de memória da struct respetiva quando no main são criadas uma variável para cada ListaUm, ListaDois e ListaTres, de nomes **lista\_um**, **lista\_dois** e **lista\_tres**, cujo contador começa em 0. São os pointers destas variáveis que são logo passados para a função menu.

Para o registo das taxas de segurança social necessários paras os cálculos salariais, os parâmetros necessários para cada cálculo foram incluídos numa struct **Taxas**.

Em relação aos cálculos salariais e à gestão dos funcionários, ambos permitem ser criados através de inputs manuais do utilizador ou através de um ficheiro de texto que esteja na pasta “UTILIZADORES” do programa. Ainda é possível verificar todos os dados para cálculos guardados e funcionários adicionados. Também se pode remover funcionários do programa, ficando estes, na verdade, marcados como inativos e não aparecendo em relatórios de funcionários.

Ao iniciar o programa, este realiza logo 5 funções para importar os ficheiros binários, guardados pelo sistema, para a memória do computador.

Em relação ao ficheiro de funcionários inseridos no sistema, a função **importar\_users\_sys** recebe o pointer de **arrayRH**. Para importar o ficheiro de cálculos salariais criados, a função **importar\_salarios\_sys** recebe o pointer de **conta**. Para importar o ficheiro de processamentos salariais, a função **importar\_salarios\_proc\_sys** recebe o pointer de **salarios**. Para importar as tabelas de IRS, são as funções **importarTabela\_um**, **importarTabela\_dois** e **importarTabela\_tres** com os pointers de **dois\_titulares**, **unico\_titular** e **nao\_casado** respetivamente. E para importar os valores da Segurança Social, a função **importarTabelaSS** recebe o pointerde **taxa**. Estas funções, exceto a última, criam um pointer de memória temporário do mesmo tipo do pointer de memória da struct apontada pelos pointers passados. São abertos os respetivos ficheiros e de seguida lido cada espaço de memória do tamanho das structs. A informação é guardada no array de memória do pointer da struct passado ao mesmo tempo que depois é criado mais um espaço nesse mesmo array de memória. No final, o pointer de memoria passado para a função passa a ser idêntico ao pointer do array temporário. Para importar a tabela da Segurança Social, é aberto o ficheiro correspondente e lidos e gravados os valores no ficheiro para os parâmetros da struct **taxa**.

Na ajuda a todos os inputs requeridos foram criadas uma série de pequenas funções que pudessem verificar limites aos valores introduzidos.

**obterInt** é uma função que define limites ao valor de inputs do tipo integer. Recebe um valor de mínimo, um valor de maximo, e uma string. Um valor é pedido enquanto que esse não se encontre entre os limites, e depois retornado esse **integer**.

**obterString** é uma função que recebe um pointer para o espaço de uma string, um valor para o seu tamanho máximo e uma string. Com a biblioteca **<ctype.h>**, é pedida o input de uma string até esta ser digna de um nome próprio, contendo apenas letras. Essa string é então guardada no espaço do pointer recebido.

**obterNum** é uma função que define limites aos valores de inputs do tipo long. Recebe um valor de mínimo, um valor de maximo, e uma string. Um valor é pedido enquanto que esse não se encontre entre os limites, e depois retornado esse long.

**est\_civilToString** e **cargoToString** são as funções que retornam a string com o nome adequado para o Estado civil ou para o cargo do funcionario, sendo que então, a primeira recebe uma variável da enumeração **Est\_Civil**, e a segunda uma variável da enumeração **Cargo**.

**estadoToString** é a função que recebe o valor do estado dos dados para cálculo salarial e retorna, dependendo desse valor, uma string com o significado.

**defineData**, que requer um valor específico para dia, mês, ano, hora, minuto ou segundo, define a hora exata do dia corrente usando a biblioteca **<time.h>**, e dependendo do valor passado retorna o parâmetro requerido.

**saberAno** pede uma string. De seguida, recorre a **obterInt** para saber um ano, onde passa um valor de mínimo, o valor de **defineData** de ano, para ano máximo, e uma string. Assim, retorna um ano possível entre tais limites.

**saberMes** pede uma string e um ano. De seguida, se o ano corresponder ao ano atual, definido com **defineData**, recorre a **obterInt** para saber um mês, onde passa um valor de mínimo, o valor de **defineData** de mês, para mês máximo (o atual), e uma string. Se não, recorre a **obterInt**, mas com o mês máximo como o 12, dezembro.

**saberDia** pede uma string, um ano e um mês, e retorna o dia de valor obtido do **obterInt** com o mínimo de dias, um texto, e o máximo de dias com **saberDiaMax**, que recebe o ano e mês em questão.

**saberDiaMax** pede um ano e mês para poder retornar o número máximo de dias que esse mês teve/tem nesse exato ano. A função tem um array de 13 posições, **meses**, com os dias dos meses em anos não bissextos, sendo que a primeira posição é 0 por não existir mês 0 e assim facilitar código. A primeira verificação é de se o ano é bissexto ou não. Dependendo disso, tem-se dois grupos de possibilidades. Se o ano não for divisível por 4, ou seja, não bissexto, verifica se o ano é o corrente e o mês também, com o **defineData**, e assim o retorno é o dia presente como máximo. Senão, o ano pode ser ou não o corrente, o retorno é o valor da posição do mês no array **meses**. Com os anos bissexto, as condições são as mesmas, exceto quando o mês é fevereiro. Aí o retorno é o valor da posição do mês, neste caso fevereiro, mas adicionando 1 dia a esse valor, passando de 28 a 29 dias.

Para criar cálculos sobre o salário de um funcionário através de inputs manuais, criou-se a função **verificacaoFuncionariosCalculo,** que recebe opointer de memória de **conta** e o pointer de **arrayRH.** Esta obtém o código do funcionário sobre o qual serão os cálculos e verifica se o funcionário com esse código existe na base de dados através da função **procurarFuncionario**. Se o funcionário existe, é iniciada a função **calcSalarial**. Se não existe, o utilizador é avisado e questionado se quer criar um funcionário com esse código. Se a resposta for afirmativa, passa para a função **addFuncionario**.

A função **calcSalarial** recebe o pointer de memória de **conta** e um código. Primeiro, define a data de saída da empresa do funcionário usando **procurarFuncionario** para descobrir esses valores. Se o funcionário ainda não saiu, usa **defineData** para criar o limite para os cálculos salariais que podem ser feitos. Depois é pedido o ano do cálculo salarial ao utilizador, enquanto que este tiver fora dos limites das datas do funcionario na empresa. A seguir, o mês é pedido tendo em atenção que se o ano for o mesmo do ano de limite máximo, o mês não pode ultrapassar o mês de limite máximo, e que se o ano for o mesmo que o ano de limite mínimo, o mês não pode ser menor que o mês de limite mínimo. Por último, pede os dias de trabalho ao longo do mês, tendo em atenção que não se podem ir excedendo em relação ao máximo de dias do mês. A verificação de se estes dias no total não ultrapassam os dias do mês é feita pela função **verificacaoDias**. Os valores são guardados no espaço de memória marcado pelo contador. Por último, pergunta se é suposto continuar com o processamento salarial dos mesmos. No caso de afirmação, chama a função **calcular**, passando como número de processamentos a fazer apenas 1.

Para criar cálculos sobre o salário de um funcionário através de ficheiros, tem-se a função **importar\_salarios\_doc**. Abre o **salarios.txt**, da pasta UTILIZADORe lê toda a sua informação, guardando os códigos e os dias de trabalho em novos espaços de memória no **conta.calculo\_array**. No entanto, a cada linha lida, ou seja, cada espaço de variáveis adicionado: se a função **procurarFuncionario** mostrar que esse código não existe nos funcionários existentes em **arrayRH.calculo\_array**, o valor da variável local **porAdd** incrementa para mostrar quantos funcionários é preciso acrescentar à base de dados; e se a função **verificaçaoDiasDoc** retornar 1, verificando que os dias apresentados no seu total não coincidem com o numero de dias do mês a ser processado, a variável **estado** daquele espaço de memoria passa a -1, ou seja invalido, e é incrementada a variável local **atencao**, que apresenta quantos conjuntos de dados para cálculos salariais são inválidos para processamento. Após fechar o ficheiro, se avisa o utilizador acerca do número de salários adicionados e quantos são inválidos. Usando, depois, o valor de **porAdd**, avisa o utilizador dos funcionários que não existem na base de dados, usando **procurarFuncionario** para mostrar os códigos em questão. No final, a função questiona se o utilizador quer ver os cálculos lidos e adicionados e, se a resposta for sim, mostra os últimos conjuntos de dados em relação a cálculos salariais adicionados. Por último, pergunta se é suposto continuar com o processamento salarial dos mesmos. No caso de afirmação, chama a função **calcular**, passando como número de processamentos a fazer o mesmo número de vezes que o programa leu uma linha.

A função **verificacaoDiasDoc** recebe os dias e o mês e ano de um salário para processar e verifica se o conjunto desses dias é equivalente ou não ao número de dias desse mês, retornando 1 se não igualar.

Para adicionar funcionários através de inputs do utilizador, é chamada a função **verificacaoAddFuncionarios**. Esta recebe o pointer de memória de **arrayRH** e pede o código do funcionário a adicionar. Primeiro verifica se este é já um funcionario existente na base de dados ou não através da função **procurarFuncionario**. Se não for existente, passa para a função **addFuncionario.**

**addFuncionario** recebe o pointer de **arrayRH** e o código e percorre as variáveis a acrescentar da struct do funcionário. Usa funções de ajuda para inputs como **obterString** para o nome, **obterNum** para o número telemóvel, **saberAno**, **saberMes**, **saberDia**, **defineData** para datas de nascimento, entrada na empresa e saída da empresa (se for o caso do funcionário), e **cleanInputBuffer**. Ao obter o estado civil, apenas pergunta o número de titulares se o funcionário for casado, senão é automático a 0. Ao saber a data de nascimento do funcionário, verifica se o ano é no mínimo de há 18 anos atras do corrente, pois o funcionário não pode ser menor e trabalhar na empresa. Se o ano for exatamente à 18 anos atras, então é verificado que o mês inserido não é superior ao mês corrente, pela mesma razão. Ao pedir o dia de nascimento, verifica se o dia não ultrapassa o dia atual, no caso de o ano ser há 18 anos atras e o mês for o corrente. Ao pedir a data de entrada na empresa, é verificado se o funcionario tinha no mínimo 18 anos. Tirando o ano de nascimento ao ano, este não pode ser menor que 18, e se este for 18, o mês tem de ser maior ou igual ao mês de nascimento, e se este for o mesmo que o mês de nascimento, o dia tem de ser maior ou igual ao dia de nascimento. Para a data de saída da empresa, se o funcionario ainda trabalhar na empresa, a data é igualada a 0s. Se já não trabalhar na empresa, é verificado que a data não é anterior à data de entrada na empresa, sendo que se o ano for equivalente, o mês não pode ser anterior ao mês de entrada, e se esses forem idênticos, então o dia não pode ser menor que o dia. O funcionário é por último tornado ativo.

Para adicionar funcionários através de um ficheiro de texto da pasta “UTILIZADOR”, a função **importar\_users\_doc** recebe o pointer de **arrayRH** e abre o **funcionarios.txt**. Do ficheiro lê toda a informação, guardando cada valor da linha numa variável do funcionário novo. Após fechar o ficheiro, o programa questiona se o utilizador quer ver toda a informação acabada de guardar e, se sim, imprime os últimos funcionários guardados no sistema, os lidos desse ficheiro.

Para editar informação sobre um funcionário é preciso primeiro verificar se esse funcionário existe. Assim **verificacaoEditarFuncionarios** recebe o pointer de **arrayRH** e pede por um código. De seguida, procura esse código nos funcionários da memoria através do **procurarFuncionario**. Assim, se o funcionário existir e tiver marcado como ativo, a sua posição no **arrayRH.funcionarios\_array** é obtida e a função permite editá-lo, mostrando a sua informação através da **infoFuncionario**. Se o funcionário estiver na base de dados mas tiver sido removido, o utilizador pode adicionar um funcionário, verificando com **verificacaoAddFuncionario** primeiro se o código a criar já existe ou não. Se o funcionario não estiver na base de dados, a função dá também a hipótese de adicionar um funcionário novo então ao sistema, através de **addFuncionario**.

Quando a função **infoFuncionario** é chamada, esta recebe o pointer de **arrayRH** e a posição do funcionário. Primeiramente mostra toda a informação guardada sobre esse funcionário e depois pergunta pelo parâmetro a mudar. Com a resposta do utilizador, **editarFuncionario** recebe o pointer de **arrayRH**, a posição do funcionário e a opção do parâmetro a editar, com o qual define que input pedir. O utilizador pode, depois, escolher mudar mais algum parâmetro também.

Ao escolher remover um funcionário, é chamada a função **removerFuncionario** com o pointer de **arrayRH.** Esta tem de primeiro certificar-se que o funcionário existe na base de dados com a **procurarFuncionario** que pergunta o código. Se o funcionário existir, o seu parâmetro de ativo passa para 0, ou seja passa a ser inativo.

Com a opção de verificar todos os funcionários na base de dados, a função **mostrarUsers**, que recebe o pointer de **arrayRH** percorre os blocos de memória a procurar os funcionários que estejam ativos e imprime-os no ecrã.

A função **procurarFuncionario**, tão necessária num sistema de armazenamento de dados, percorre o array de memória através do pointer e verifica se em algum funcionário tem o mesmo código que lhe foi passado, retornando a sua posição se encontrar uma igualdade de códigos, ou retornando -1 se não.

A função **calcular** é a função que trata dos processamentos dos cálculos salariais. O número de salários a processar depende do desejado do utilizador, podendo ser todos ou os últimos x equivalentes ao número de dados para cálculos carregados, passados como argumento da função. Em primeiro lugar, verifica se os dados do espaço da memoria de **conta.calculo\_array** já foram por acaso processados. Se não, continua para descobrir a posição do funcionário em questão no array de memoria **arrayRH.funcionarios\_array**, através de **procurarFuncionario**. Aqui começa com os cálculos, indo buscar os valores à posição corretas em **conta.calculo\_array**. Calcula as horas completas trabalhadas nesse mês, o salário pelos dias completos, pelos dias meios completos e pelos dias de fim de semana, somando-os todos com o bónus, conseguido através de saberBonus, para obter o vencimento ilíquido. Calcula também o valor total de subsídio de alimentação, o valor a descontar de Segurança Social para o funcionário com saberSS(1), para saber a percentagem, e para a empresa com saberSS(0), o valor do IRS com **saberIRS**, e por último o vencimento ilíquido somando o vencimento líquido com o total de subsídio de alimentação e descontando as respetivas partes para Segurança social do funcionário e para IRS. De seguida guarda todos estes valores no espaço disponível de **salarios.calculo\_array** e muda o estado dos dados para calculo em **conta.calculo\_array** para processado com 1. Por último, é adicionado ao ficheiro RELATORIO.txt toda a informação acabada de guardar sobre o processamento salarial, especificando também o nome do funcionario vindo de **arrayRH.funcionarios\_array**, e o mês e ano vindos de **conta.calculo\_array**. Se houve processamentos, por último a função pergunta se o utilizador deseja ver os dados finais desses mesmos.

É possível também escolher ver os salários que ainda estão por processar e os salários que já foram processados. Para mostrar os processados, é percorrido o array de memoria **salarios.calculo\_array**, imprimindo todos os seus dados. No entanto, para mostrar os não processados, é percorrido o array de memoria **conta.calculo\_array** procurando os espaços cujo estado é diferente de 1 ou seja, não foi processado ainda, e imprime os dados desses salários.

Tanto na escolha do menu de gestão de tabelas de descontos para o IRS ou das taxas da segurança social, tem-se a escolha de alterar critérios ou mostrar as tabelas com os valores todos. Para alterar os critérios de IRS, é iniciada a função **alterarCriterioIRS**, passando os pointers de memória de **dois\_titulares**, **unico\_titular** e **nao\_casado**. Aqui é perguntado ao utilizador e guardado o valor da tabela que se quer alterada, o vencimento da linha em questão, o número de filhos e o novo valor para substituir. O valor da tabela, do vencimento e os pointers de **dois\_titulares**, **unico\_titular** e **nao\_casado** são passados para a **verificarVencimento** poder certificar-se de que existe esse valor de vencimento na tabela de IRS escolhida. Agora dependendo do valor da tabela, é percorrido o array de memória de cada uma, escolhendo a linha do vencimento idêntico ao valor aceite, e é substituído o valor na coluna de número de filhos correta.

Para serem mostrados todos os valores de uma tabela de descontos de IRS, a função **mostrarTabelas** recebe os pointers de **dois\_titulares**, **unico\_titular** e **nao\_casado** e dependendo da escolha da tabela percorre o array de memória referente a cada tabela, **array\_um->Dois\_titulares\_array[i]**, **array\_dois->Unico\_titular\_array[i]**, **array\_tres->Não\_casado\_array[i]**, imprimindo as linhas com o vencimento e os valores para cada número de filhos.

**verificarVencimento** avalia o valor passado por tabela e percorre o array de memória da tabela escolhida procurando um vencimento que seja idêntico ao passado nos argumentos.

A alteração de taxas de Segurança Social decorre como o início em relação aos valores de IRS, em **alterarCriterioSS**, a perguntar que categoria deseja alterar, e o seu novo valor, que substitui o antigo valor na categoria escolhida. É depois chamada a função **mostrarTaxas** com o pointer de memória de **taxa**. O sistema pede a categoria cujo valor vai ser trocado e depois o próprio valor, mudando então depois na categoria correta da struct do tipo **Taxas**.

**mostrarTaxas** recebe o pointer de **taxa** e imprime cada valor da struct.

Tanto quando o utilizador escolhe guardar toda a informação pelo menu, ou quer sair do programa e é perguntado se deseja guardar os dados, é chamada a função **guardar**. Esta recebe os pointers para as várias structs **arrayRH**, **conta**, **dois\_titulares**, **unico\_titular**, **nao\_casado**, **taxa** e **salarios**. A função cria um pointer para abrir cada um dos ficheiros relacionados com os arrays de memória, users.bin, salários.bin, SALARIOS\_PROCESSADOS\_bin, DEPENDENTE\_CASADO\_DOIS\_TITULARES\_bin, DEPENDENTE\_CASADO\_UNICO\_TITULAR\_bin, DEPENDENTE\_NAO\_CASADO\_bin e TAXAS.txt, e reescreve em cada um toda a informação existente nos arrays de memória, por cada espaço de uma struct.

A opção de apagar os dados em memória chama **limpar\_memoria**, que primeiro assegura o utilizador da escolha. Abre os tres ficheiros relacionados com os funcionários, com os dados de cálculos e com os processamentos salariais, para escrita, limpando tudo no ficheiro. Este é depois fechado.

A função **saberSS** retorna o valor de Segurança Social necessário para os cálculos salariais. Precisa de receber um valor para entidade, para saber se se trata do cálculo para o trabalhador ou para a entidade empregadora. A seguir, dependendo do valor de entidade, vai buscar o cargo do funcionário em questão, ao o encontrar na memoria dos funcionários, ao passar o código recebido no início para **procurarFuncionario**. Dependendo da posição do cargo na enumeração **Cargo**, a função retorna então o valor requerido da struct **Taxas**.

A função **saberIRS** retorna o valor de IRS necessário para os cálculos salariais. Precisa de receber o vencimento do funcionário e o seu código. Primeiro usa **procurarFuncionario**, com o código e pointer para o array de memória **arrayRH.funcionarios\_array** passados no início da função, para obter a posição do funcionário na memoria. Com essa posição definida, o programa sabe então o valor de titulares do funcionário, o qual usa para decidir a que array de memoria das tabelas de IRS deve ir buscar o valor. O programa percorre então o array de memória correto, procurando um espaço cujo vencimento seja equivalente ao vencimento do funcionário, ou onde este se encontre entre os vencimentos de dois espaços consecutivos. Quando encontra uma dessas condições, a função retorna o valor do IRS do espaço correto dependendo do número de filhos do funcionário.

**SaberBonus** retorna a percentagem do bónus de um funcionário. Esta função recebe um dia\_max, que define o número máximo de dias que o mês do cálculo em questão pode ter, e um num\_salarios e x que definem a posição no array de memoria **conta.calculo\_array** do processamento a ser realizado. É primeiro calculado o número de dias trabalhados pelo funcionário. Depois são calculadas as percentagens de tempo e idade, relacionando-se com a antiguidade do funcionário a trabalhar na empresa e com a idade deste possível na empresa, respetivamente. Para o **tempo**, multiplica-se os 0.53 pelo número a trabalhar/trabalhados na empresa, recorrendo ao **defineData** do ano. Para a **idade**, multiplica-se os 0.26 pelo número de anos de idade acima do mínimo na empresa, 18, recorrendo também ao **defineData** do ano. Por último é calculada a percentagem do bónus dos **dias** trabalhados para além do obrigatório por mês. Desde logo, verifica-se se o número de dias trabalhados (**dias\_trab**) é menor ou igual a 22, pois então essa parte do bónus fica a 0. Se for maior que 22, tem de se calcular quanto, para depois multiplicar pelos 12.5 divididos pelo máximo de dias extra que podem ser trabalhados nesse mês. A percentagem do bónus é então conseguida ao somar as três partes e a dividi-las por 100.

A percentagem de **bónus** foi decidida para que pudesse ser no máximo 50% do vencimento ilíquido. Esses 50% seriam divididos em 3, sendo que então, no máximo, 25 seriam em relação à antiguidade do funcionário na empresa, 12.5 da idade possível do funcionário na empresa, e outros 12.5 seriam dos dias trabalhados a mais dos obrigatórios por mês (decididos como 22).

O cálculo dos 25% da **antiguidade** baseiam-se no máximo de anos que um funcionário pode trabalhar na empresa, ou seja dos 18 anos (idade mínima de entrada na empresa) aos 65 anos (idade de reforma), que são 47 anos. Dividindo então os 25 por 47 anos, equivale a 0.53% por cada ano a trabalhar na empresa. Para se obter a parte em relação à antiguidade do funcionário basta assim multiplicar os anos trabalhados por 0.53.

O cálculo dos 12.5% da **idade** do funcionário baseiam-se na idade que um funcionário pode ter na empresa, ou seja, como só se pode começar a trabalhar na empresa aos 18 e tem de se sair aos 65, esta percentagem vai também ser dividida por 47 anos, dando 0.26 por ano de idade acima dos 18.

O cálculo dos 12.5% dos **dias extra** trabalhados para além dos 22 obrigatórios por mês vai dependendo do número de dias do mês em questão. Sabendo o número máximo de dias extra que podem ser trabalhados nesse mês, retirando 22 aos dias totais do mês, divide-se os 12.5 por esses dias. Esse número é então multiplicado pelo número de dias que o funcionário trabalhou extra. Esses dias são calculados retirando-se 22 aos dias que trabalhou nesse mês, que são a soma dos dias completos, meios completos e de fim de semana trabalhados.

Por fim, são somadas essas tres partes, dando a percentagem de bónus final, que com 65 anos de idade a trabalhar na empresa à 45 anos e trabalhando os dias todos de um mês, seria 50%.

**FUNCIONALIDADES PROPOSTAS**

Para as 5 funcionalidades propostas, o grupo escolheu apresentar 8 listagens que considerou mais práticas e necessárias. Pelo **menu\_listagens** temos a possibilidade de mostrar os funcionários que já não trabalham na empresa, o gasto total da empresa com os cálculos salariais processados, o gasto total da empresa em impostos, os funcionários removidos pelo utilizador, e um conjunto de 3 listagens de dados guardados no sistema em ficheiro, os funcionários, os dados para cálculos, e os cálculos salariais processados.

Para listar os **funcionários que já não trabalham** na empresa, o sistema percorre o array de memoria **arrayRH.funcionarios\_array**, procurando os funcionários cuja variável do ano de saída da empresa seja diferente a 0, sendo que 0 é o valor para os que continuam na empresa, imprimindo depois de cada desigualdade o código do funcionario que já saiu.

Para apresentar o **gasto total** pela empresa, é percorrido o array de memoria **salarios.calculo\_array**, somando todos os valores de encargo total da empresa por cada processamento salarial guardado, imprimindo no fim essa soma.

Para apresentar o **gasto da empresa em impostos** com os processamentos salariais realizados, é percorrido o array de memoria **salarios.calculo\_array**, somando todos os valores de Segurança Social por parte da entidade patronal por cada processamento salarial realizado, imprimindo no fim essa soma.

Para listar os **funcionários removidos** do programa, o sistema percorre o array de memoria **arrayRH.funcionarios\_array**, procurando os funcionários cuja variável ativo seja 0, pois 0 é o valor para os que não tao ativos no sistema, imprimindo depois de cada igualdade o código do funcionario inativo no sistema.

Para criação das três listas em ficheiro, **criarFicheirosUser** abre para cada opção o ficheiro correspondente, FUNCIONARIOS\_TEXTO.txt, SALARIOS\_TEXTO.txt, CALCULOS\_TEXTO.txt, e em cada um imprime a informação por cada espaço do array de memoria certo, **arrayRH.funcionarios\_array**, **conta.calculo\_array**, **salarios.calculo\_array**.