CONCURSO P1

1. Preliminares

Em **Programação Imperativa** vamos aprender a programar com a Linguagem de Programação C. Em C, um programa é composto por um conjunto de funções. Uma função é um conjunto de comandos agrupados num bloco que recebe um nome e através do qual pode ser evocado. Existe, no entanto, uma função especial, a função **main**. Esta função **main** é especial porque é nela que o programa inicia todas as operações quando é iniciado. Inversamente, quando a função **main** esgota as suas instruções, o programa termina.

A Linguagem C, apesar de antiga, picuinhas e talvez chata, comparada com outras linguagens de programação mais atuais, continua a ser a linguagem mais estruturada e que melhor demonstra todas as áreas de aprendizagem de programação, pois quem sabe programar em C, facilmente programa noutra linguagem qualquer!

Para programar com C, precisamos de um computador, um editor de texto e um compilador que transforme os nossos programas-fonte em executáveis no nosso computador.

O editor de texto "oficial" em **Programação Imperativa** é o Sublime Text (https://www.sublimetext.com/download). É muito popular e tem a vantagem de poder ser utilizado em qualquer plataforma (Windows, MacOS ou Linux). Se ainda não instalou, deverá fazê-lo.

O compilador "oficial" em **Programação Imperativa** é o **gcc**. Também é multiplataforma e em certas "distribuições" de Linux vem logo instalado (https://osdn.net/projects/mingw/releases). Deverão instala-lo para que possam iniciar o desenvolvimento. (Ver manual de instalação na tutoria eletrónica https://tutoria.ualg.pt , na UC de Programação Imperativa).

2. Diretoria de Trabalho

Deve criar uma diretoria de trabalho EngINF para colocar todas as cadeiras do curso e uma subdiretoria para os trabalhos desta cadeira, PI2022. Dentro de PI2022 deverá criar dois subdiretórios, sources (onde armazenará todos os programas efetuados e entregues) e work (onde armazenará os ficheiros de dados que os nossos programas usarão. Por enquanto fica vazia). Poderá criar outras diretorias, como aulas, para as apresentações das aulas teóricas e docs para os textos fornecidos pelos professores, como por exemplo este documento.

3. Vamos iniciar a programação – Programa A no Mooshak

Vamos iniciar com um exemplo bem simples, que estudamos na aula teórica 1: a soma de dois números.

```c
#include <stdio.h>

int main()
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = x + y;
    printf("%d\n", z);
    return 0;
}
```

O programa tem uma única função, a função **main**, onde estão expressos todos os cálculos e a sua impressão do resultado para o ecrã.

3.1. Editor de texto

Deverá abrir o **Sublime Text** para editar o texto. Transcreva o texto no editor e assim que estiver o texto todo escrito, guarde com o nome **sum.c** dentro da diretoria **sources**. Tenha atenção em guardar com a extensão " **.c** " para que o editor possa reconhecer o formato.

3.2. Compilação

Agora depois de transcrição, vamos compilar o programa para que possamos torna-lo um executável e podermos corre-lo. A compilação, conforme foi mencionado na aula teórica, serve para criar um binário executável, ou seja, transformar as nossas instruções em instruções que o computador entenda e que possam ser executadas.

Para poder compilar deve abrir uma janela de comando ou consola e dirigir-se para a diretoria **sources** onde se encontra o ficheiro **sum.c .**

Em Windows é muito simples, ir ao botão iniciar, clicar nele e depois escrever **cmd.** Depois dirigir-se ao local onde criou a pasta **source**, por exemplo cd EngINF\PI2022\sources, se estiver dentro de documents do seu utilizador.

No MacOS, é parecido: no **Finder**, seleciona-se a pasta onde queremos abrir a janela e dá-se **Finder->Services->New Terminal at Folder**.

No Linux, abrir uma consola e escrever cd /EngINF/PI2022/sources.

Sugestão: Coloque no seu desktop os 3 links que necessita, o do **Sublime Text**, a **Janela de comando ou consola**, e a pasta **PI2022**, para ser mais fácil aceder a qualquer uma delas.

Feito este preambulo, vamos finalmente compilar o programa. Para tal, conforme referido anteriormente, deverá ter a **consola ou janela de comando aberta** na pasta **sources**, e dar o seguinte comando na consola:

gcc -Wall sum.c

Se surgir uma mensagem de erro indicando que o sistema operativo não reconhece o **gcc**, isso significa que não está instalado ou que não se encontra disponível na **Path**, pelo que deverão rever o manual de instalação que se encontra na tutoria eletrónica e seguir os passos da instalação.

Se surgir a mensagem que "os ficheiros não existem", significa que o **gcc** encontra-se bem instalado, mas que não se encontra na diretoria onde criou o ficheiro **sum.c** que queria compilar, pelo que deverá ir para a diretoria correta ou então não deu o nome correto ao ficheiro e deverá modificar o nome do mesmo.

Se a compilação não der qualquer erro, surgirá na mesma diretoria um ficheiro denominado **a.exe** em Windows w **a.out** em Linux ou MacOS, sendo esse o ficheiro executável criado.

3.3. Execução

Para executar o programa, invocamos na linha de comandos ou consola o ficheiro **a.exe** ou **a.out,** consoante o sistema operativo, Windows ou Linux/MacOS.

Eis a transcrição de uma sessão em Windows, com três ensaios:

```
>a.exe
5 7
12
>a.exe
100 200
300
>a.exe
571 219
790
>
```

O sinal > que surge na transcrição da sessão é o prompt, que indica que a consola está a aguardar o input do utilizador.

A transcrição em MacOS ou Linux:

```
$ ./a.out
5 7
```

```
12
$ ./a.out
100 200
300
$ ./a.out
571 219
790
$
```

Neste caso o prompt é $.

3.4. Submissão

Alguns programas que desenvolverá no âmbito desta cadeira, serão avaliados automaticamente pelo Mooshak. Nesses casos, o professor terá colocado nessa plataforma vários casos de teste. O programa correrá com cada um dos casos de teste. Se em todos os casos o resultado do seu programa for o esperado, o seu programa será imediatamente aceite, caso contrário, ele não aceitará e poderá dar várias respostas, como Wrong Answer, Presentation Error, etc. e ai terá de rever o seu código.

Para praticar este funcionamento, submetamos o programa **sum.c** no concurso PI_2223_P1 do Mooshak. Começamos por fazer login no Mooshak, escolhendo o concurso PI_2223_P1.

Constatará que o concurso tem XXXX problemas. O problema A é o problema que deu origem ao programa **sum.c**. Os outros problemas virão a seguir.

Use a interface do Mooshak para submeter o programa no problema A e verificará que o seu programa está **Accepted**.

Agora que foi aceite, não lhe mexa mais! Se precisar de modificá-lo, trabalhe numa cópia com outro nome. Em todo o caso, se perder o ficheiro de um programa aceite, pode sempre vir buscá-lo, novamente ao Mooshak.

3.5. Analisando o programa

Este nosso primeiro programa, na verdade foi "oferecido" já pronto. Para ganhar familiaridade com a linguagem, modifique o programa (sempre a partir de uma cópia) de maneira a, por exemplo, somar 3 números, realizar outras operações aritméticas, etc. Experimente também mudar o programa, de maneira a forçar alguns erros depois de compilar. Leia as mensagens de erro e interprete-as. Há de reparar que nem sempre as mensagens de erro descrevem o erro de uma maneira que se entende imediatamente. Por exemplo, apague a diretiva **#include** na primeira linha. Que acontece? Apague, o **void** na lista de argumentos da função **main**? Apague o **return 0**; na função **main**. Acrescente um **return 0**;. Omita o ponto e vírgula no **return 0;**. Apague o **&** dentro do scanf. Apague a linha **int x;**. Apague o sinal **=** na linha **z = x + y;**. No printf, escreva **%f**, em vez de **%d**. E depois **%o**. E também **%x**. E ainda **%y**. Omita o **\n**. Escreva **som** em vez de **sum**, na chamada da função. Experimente escrever **Int** em vez de **int**. Apague o **int** antes do **z**, etc. Em cada caso, interprete a mensagem de erro do compilador, se houver. Às vezes há erros — em inglês, errors — e às vezes também há avisos — em inglês warnings. Nós só ficamos satisfeitos quando não houver nem erros, nem avisos. Em geral, se executarmos um programa que não tem erros de compilação, mas tem avisos, as coisas até podem correr bem, mas não é nada recomendável. Só experimentaremos um programa depois de ter eliminado todos os erros de compilação e todos os avisos. Note que se executar um programa depois de uma compilação com erros, na verdade estará a correr a anterior versão do programa que terá sido compilada sem erros, e não aquela que tinha erros. Em resumo, será este o ambiente de trabalho nos primeiros tempos: uma janela com o editor, que usamos para escrever o programa, e uma janela de comando, que usamos para compilar e correr o programa. Episodicamente, uma janela com o Mooshak, para submeter.

4. Segundo Problema, soma! Problema B no Mooshak.

O primeiro problema, conforme mencionado anteriormente, foi oferecido e, como tal, não constituiu um desafio. Nas aulas teóricas vimos um outro programa para somar números:

```c
#include<stdio.h>

int sum(int x, int y)
{
    return y==0 ? x : sum(x+1, y-1);
}

int main (void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = sum(x, y);
    printf("%d\n", z);
    return 0;
}
```

Neste caso, supusemos, como exercício, que o C não "sabia" somar dois números, mas que apenas sabia somar uma unidade e subtrair uma unidade de cada vez.

Vamos agora, continuando o exercício, exagerar naquilo que o C não sabe fazer. Leia com atenção, porque pode parecer estranho. Suponhamos que o C não tem os operadores aritméticos, isto é, não tem o sinal **+**, nem **-**, nem *****, nem **/**. Também não tem numerais, exceto 0 e 1. Isto é, os únicos numerais que o compilador de C "entende" são **0** e **1**. Se, por exemplo, aparecer o numeral **18** no programa, o compilador não sabe de que se trata. E também não tem operadores de comparação **==**, **<**, etc. Por hipótese ainda, para efeitos deste exercício, o C traz quatro funções aritméticas que podemos usar nos nossos programas. São as seguintes: **succ**, **pred**, **is_zero**, **is_pos**.

A função **succ** calcula o sucessor de um número inteiro. Por exemplo, **succ(0)** vale 1, **succ(45)** vale 46, **succ(-18)** vale -17. A função **pred** calcula o predecessor de um número inteiro. Por exemplo, **pred(1)** vale 0, **pred(62)** vale 61, **pred(-32)** vale -33.

A função **is_zero** dá 1, se o argumento for 0, e dá 0 em todos os outros casos. Por exemplo, **is_zero(0)** vale 1, **is_zero(1)** vale 0, **is_zero(14**) vale 0, **is_zero(-15)** vale 0.

A função **is_pos** dá 1, se o argumento for um número positivo ou 0, e dá 0 em todos os outros casos. Por exemplo, **is_pos(0)** vale 1, **is_pos(1)** vale 1, **is_pos(26)** vale 1, **is_pos(-15)** vale 0. O exercício agora é reescrever a função **sum** do programa acima, respeitando estas "regras do

jogo", isto é, sem usar os operadores aritméticos, sem usar a operadores de comparação e sem usar numerais, exceto 0 e 1 (se for caso disso).

Realize isso num novo ficheiro, por exemplo **sumb.c**, mantendo a linha **#include** e a função **main** e modificando o corpo da função **sum**, utilizando as funções acima descritas, para respeitar as regras do exercício.

Adicionalmente, a seguir à linha **#include**, insira a esta outra linha:

const char *author = "Your name here";

E coloque o seu nome onde está **Your name here**. Faça isso sempre, nos programas que escrever.

Acrescente também as seguintes quatro linhas, a seguir à linha com o seu nome:

int succ(int x);

int pred(int x);

int is_zero(int x);

int is_pos(int x);

Aquelas quatro funções constituem uma mini-biblioteca, que chamamos **minic**, que, não sendo "standard" do C, temos de ser nós a gerir. O seu programa ficará com o seguinte aspeto:

```c
#include <stdio.h>

const char *author = "Your name here";

int succ(int x);
int pred(int x);
int is_zero(int x);
int is_pos(int x);

int sum(int x, int y)
{
    // TO DO: Coloque aqui o seu código
}

int main(void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = sum(x, y);
    printf("%d\n", z);
    return 0;
}
```

Falta escrever o código da função **sum**, que substituirá a linha que inicia como **//TO DO:**

Quando terminar o seu código e a nova função **sum** estiver concluída, compile o seu programa. No Windows dará um erro idêntico ao abaixo descrito, assim como nos outros sistemas operativos:

```
urces\Aula 2>gcc -Wall Aula2_somab.c
c:/mingw/bin/../lib/gcc/mingw32/9.2.0/../../../../mingw32/bin/ld.exe: C:\Users\JDGUER~1\AppData\Local\Te
mp\ccorTLFt.o:Aula2_somab.c:(.text+0xe): undefined reference to `is_zero'
c:/mingw/bin/../lib/gcc/mingw32/9.2.0/../../../../mingw32/bin/ld.exe: C:\Users\JDGUER~1\AppData\Local\Te
mp\ccorTLFt.o:Aula2_somab.c:(.text+0x1d): undefined reference to `pred'
c:/mingw/bin/../lib/gcc/mingw32/9.2.0/../../../../mingw32/bin/ld.exe: C:\Users\JDGUER~1\AppData\Local\Te
mp\ccorTLFt.o:Aula2_somab.c:(.text+0x2a): undefined reference to `succ'
collect2.exe: error: ld returned 1 exit status
```

O compilador assinala que há 3 referências não definidas**, is_zero**, **pred** e **succ**. Na verdade, falta mencionar ao compilador para utilizar a biblioteca **minic.c**. Para tal, temos de acrescentar a biblioteca na diretoria **sources**, bilbioteca que pode descarregar da tutoria eletrónica e colocar na referida diretoria. Seguidamente, compile utilizando o seguinte comando:

gcc – Wall sumb.c minic.c -O2

Se compilar sem erros, experimente correr o programa e faça os seguintes testes:

```
urces\Aula 2>gcc -Wall Aula2_somab.c minic.c -O2

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>a
6 11
17

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>a
78 0
78

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>a
1000000000
1000000000
2000000000

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>a
1 -1
0

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>a
-23 10
-13

C:\Users\jdguerreiro\OneDrive - Universidade do Al
urces\Aula 2>
```

Observe com atenção os resultados, teste com outros números até ficar confiante que o seu programa calcula bem. Nessa altura, submeta-o no Mooshak, **concurso PI_2223_P1, problema B.**

Note! Se fizer batota, usando no programa operadores aritméticos ou numerais que não 0 ou 1, o seu programa até pode ser aceite pelo Mooshak, mas os professores invalidá-lo-ão depois.

Se ficou curioso com a opção de compilação, **-O2**, no fim do comando para compilar, experimente de novo, sem usar **-O2** e veja as diferenças. 😊

5. Terceiro Problema, Diferença – Programa C no Mooshak

Já sabemos somar, recorrendo apenas às funções **succ, pred** e **is_pos**. O exercício seguinte é um programa para subtrair. Mais precisamente, queremos uma função **difference**, com dois argumentos de tipo **int** cujo resultado é a diferença "primeiro argumento menos segundo argumento". E, como exercício, usamos as mesmas "regras do jogo".

Resolva este exercício num terceiro programa, de nome **difference.c.** O comando de compilação é idêntico ao anterior, utilizando **difference.c** em vez de **sumb.c**:

gcc -Wall difference.c minic.c -O2

Submeta no Mooshak no **problema C**.

6. Quarto problema, menor ou igual – Problema D no Mooshak

O quarto exercício é para programar o operador de comparação menor ou igual, usando as mesmas regras anteriores, num novo ficheiro atribuindo o nome **less_equal.c**.

Em particular, a função que resolve o problema deve chamar-se **less_equal**. O resultado deve ser 1 (um) quando o primeiro argumento for menor ou igual ao segundo e deve ser 0 (zero) quando não for.

Note bem, que além das quatro funções da biblioteca **minic.c**, o seu programa pode também usar as funções que já programou anteriormente, **sum** e **difference**. Se precisar de alguma destas, copie-a para o seu novo programa, colocando-a antes da nova função **less_equal.**

Submeta no Mooshak, **problema D**.

7. Quinto problema, dobro – Problema E no Mooshak

Queremos agora uma função **twice**, tal que **twice(x)** é o dobro de **x**. Por exemplo, **twice(3)** vale 6, **twice(50)** vale 100, **twice(-1)** vale -2. Programe num novo ficheiro **twice.c**. Note que temos agora uma função unária, isto é, uma função com um só argumento. Por isso, a função **main** não precisa da variável **y** e a instrução **scanf** fica mais simples. Submeta no Mooshak, problema E.

8. Sexto problema, metade – Problema F no Mooshak

Agora um problema mais subtil: calcular metade de um número inteiro positivo ou zero. Será a função **half**. No caso de o argumento ser ímpar, queremos o quociente da divisão por 2 arredondado para baixo. Por exemplo, **half(10)** vale 5, **half(17)** vale 8. Note que só estamos interessados em números que não sejam negativos. Se o argumento for um número negativo, comportamento da função não está definido. Programe num novo ficheiro **half.c**. Submeta no Mooshak, **problema F**, sabendo que em todos os testes, os números lidos são números não negativos.

9. Sétimo problema, máximo, Problema G no Mooshak

As duas funções anteriores, **twice** e **half** são muito importantes em programação, e são fornecidas diretamente pelo hardware. Na prática, não é preciso programá-las, como aqui fizemos, por exercício. Outra função muito usada é a que, dados dois números, calcula o maior dos dois. Chama-se **max**. Por exemplo, **max(4, 7**) vale 7, **max(-3, -5)** vale -3, **max(8, 8)** vale 8. Programe num novo ficheiro **max.c**. Submeta no Mooshak, **problema G**.

10. Oitavo problema, potencia de base 2, Problema H no Mooshak

Oitavo problema: potência de base 2 As potências de base 2 são muito importantes em programação. Em breve, você conhecê-las-á todas de cor, desde 2^0 até 2^{31} 😎. Para se ir habituando, use o programa que vai escrever agora, para calcular potências de 2. Queremos uma função unária, **power_of_2**, tal que **power_of_2(x)** é o valor de 2 elevado a x. Por exemplo, **power_of_2(3)** vale 8**, power_of_2(10)** vale 1024.

Programe num novo ficheiro **power_of_2.c**.

Dica: 2 elevado a 5 é o dobro de 2 elevado a 4; 2 elevado a 4 é o dobro de 2 elevado a 3; …

Submeta no Mooshak, **problema H**.

Fim