

## Trabalho Prático – Parte 1

O objetivo deste trabalho é que o aluno aprenda a usar **estruturas de dados do tipo matriz, subalgoritmos (funções), macros do pré-processador, bibliotecas diferentes das utilizadas até então e variáveis globais de forma correta** em algoritmos, bem como implementá-los corretamente em programas.

### Especificação da primeira etapa:

**Problema:** Implementar um jogo em um grid 2D que possui diversos personagens diferentes, em que todos estes personagens, exceto o controlado pelo jogador, executam suas respectivas ações somente após o jogador movimentar seu avatar, e permanecem esperando até que o jogador execute o próximo movimento.

O jogador tem como objetivo conseguir o maior número de pontos antes do fim do jogo.

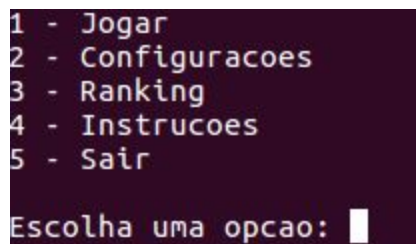
As formas de obtenção de pontos, comportamento de inimigos, controle do avatar e fim do jogo, serão especificadas posteriormente.

### Detalhamento do problema:

1. A unidade de entrada deste trabalho deverá ser o teclado para inserção dos comandos de manipulação das sequências. A unidade de saída será o monitor de vídeo.

Para facilitar o manuseio do programa, deverão ser dadas mensagens explicativas em alguns casos conforme indicado posteriormente nesta especificação.

Inicialmente deve ser mostrada uma tela de boas vindas com o nome do jogo (escolhido pelo desenvolvedor) até que o usuário tecle <enter>, em seguida deve ser apresentada ao usuário a seguinte tela, até que o usuário digite o número da opção que deseja e tecle <enter>. Conforme mostrado na figura abaixo.



```
1 - Jogar
2 - Configuracoes
3 - Ranking
4 - Instrucoes
5 - Sair

Escolha uma opcao: █
```

Inicialmente, nesta etapa, serão implementadas somente as opções de “Jogar” e “Instruções”. Ou seja, “Configurações” e “Ranking” serão implementados na Parte 2 do trabalho.

## Arena:

A arena do jogo será representada por um grid 2D de tamanho 20x20, como a mostrada abaixo:

```
Pontos : 25      Jogadas restantes : 390
.....BB.....O
.....BBB.....
.....
.....
.....
.....BB...X...
.....BB.....
.....B.....
.....
.....X
.....
.....C.....
.....
.....Q.....
.....B.....
.....BB.....
.....BB.....
.....
.....O.....
```

Em que os espaços livres para movimento são representados pelo caractere '.' e os demais caracteres representam diferentes personagens.

## Personagens:

**C** – Será o avatar do jogador, ele poder ser movimentado para quatro direções, cima, baixo, esquerda e direita, e não pode ficar parado durante um turno, caso o jogador digite uma tecla inválida deverá ser mostrada uma mensagem de erro ou realizado um movimento aleatório como punição.

**X** – Inimigo agressivo, se movimenta no intuito de atacar o 'C', caso ele alcance o jogador o jogo deve ser encerrado com uma mensagem de "GAME OVER" e a pontuação final. O 'X' deve se movimentar perseguindo o 'C' por alguns turnos e aleatoriamente em outros.

**B** – Inimigo passivo, se movimenta deixando um rastro de si mesmo no caminho criando assim uma barreira que não pode ser atravessada por outros personagens, este rastro deve conter um tamanho limitado de no mínimo 2 e o máximo podendo ser definido pelo programador. Este limite máximo poderá ser alterado posteriormente nas "Configurações". Seu movimento é sempre aleatório.

**O** – Ponto, fica parado durante todo o jogo, se o jogador conseguir pega-lo são obtidos alguns pontos.

**Q** – Inimigo bomba, aparece aleatoriamente no tabuleiro durante o jogo a cada certo período de tempo, e após um certo tempo explode matando o jogador caso ele esteja a área de explosão, a reação do jogador ao ser atingido pela bomba deve ser a mesma de ser atingido pelo 'X'. A área de explosão da bomba deve ser representada por outro caractere, '#' por exemplo. Esta explosão deverá atingir no mínimo 8 posições do grid do jogo, e o limite máximo pode ser definido pelo programador. Deverá também ser observado os limites do grid. O tamanho máximo de alcance da área da explosão poderá ser alterada posteriormente nas configurações. O formato da área atingida (ex: cruz, \*, X) poderá ser definido pelo programador e, a princípio, não faz parte das configurações.

## **Inicialização da partida:**

A inicialização da partida deve ser executada quando o usuário escolher a opção “Jogar” no menu inicial, esta inicialização consiste em sortear os locais onde os personagens serão posicionados e a definição do limite de jogadas que o jogador pode fazer até o fim do jogo.

## **Formas de se encerrar uma partida:**

Existem diferentes maneiras de se encerrar uma partida. A seguir cada uma delas serão citadas e explicadas detalhadamente.

**Jogadas esgotadas:** Uma forma de se encerrar uma partida é quando o jogador já executou o número máximo de jogadas permitidas, pré-estabelecido no início da partida, quando este número chega a 0 deve ser encerrada a partida.

**Explodido por uma bomba:** Caso o personagem fique sobre a região de alcance da bomba quando esta explodir, o jogador perderá a partida e o jogo deverá ser encerrado .

**Ataque de inimigo:** Se um inimigo se mover para a posição em que o jogador está o jogo deverá ser encerrado.

**Vencendo o jogo:** Caso o jogador consiga obter todos os pontos (opcional: e matar todos os inimigos) o jogo deve ser encerrado.

Para todas as formas de encerrar o jogo deve ser mostrada uma mensagem de “GAME OVER”, motivo do fim do jogo e a pontuação final do jogador.

**ATENÇÃO:** após cada atualização, você deve limpar a tela e mostrar a matriz na mesma posição anterior. Isso dará ao usuário a impressão de movimento.

## **Instruções**

A tela de instruções deve mostrar um texto explicativo sobre o funcionamento do jogo, as ações que o jogador pode fazer, o comportamento dos inimigos, etc.

## **Encerramento**

Ao encerrar o jogo deve ser mostrada uma tela de despedida.

(veja APÊNDICE nas próximas páginas!)

## APÊNDICE:

Neste trabalho você DEVE modularizar o seu programa criando funções. Conforme já dito em sala de aula, funções tornam o seu programa mais legível, mais fácil de ser entendido, e mais curto, já que evita a duplicação de código para uma mesma ação. Será de total responsabilidade do programador decidir quais funções serão criadas, definindo seus tipos, conteúdo e funcionalidade.

**Novos cabeçalhos, Macros e variáveis globais:** Neste trabalho, será necessário o uso de macros do pré-processador, os cabeçalhos *stdlib.h* e *time.h* e **variáveis globais**.

### 1) Das macros do pré-processador

As diretivas do pré-processador, isto é, os comandos iniciados com o caractere # que incluímos no início de nossos arquivos de código-fonte são meios de realizar modificações em um programa em tempo de compilação. Cada uma das diretivas instrui o pré-processador a realizar uma modificação no código-fonte, antes que o compilador vá, de fato, gerar o arquivo executável de um programa.

Até agora, conhecemos apenas a diretiva *#include*. Esta diretiva diz ao pré-processador para, literalmente, substituir sua declaração por todo o código que está dentro do arquivo especificado.

Neste trabalho, utilizaremos uma nova diretiva: o *#define*. Esta diretiva diz ao pré-processador para, literalmente, substituir a **macro definida** em todos os lugares que ele aparecer no código-fonte por sua **expansão**. Por exemplo: do código a seguir, será compilada a instrução *printf("%d\n", 50)* e não a instrução *printf("%d\n", MAX\_JOGOS)*. Além do mais, o compilador não saberia como compilar esta última instrução, já que não declaramos nenhuma variável com o identificador *MAX\_JOGOS*. Vejam:

```
#define MAX_JOGOS 50
...
printf("%d\n", MAX_JOGOS);
...
```

A outra maneira de se definir macros é definir macros **function-like**. Podemos encarar este tipo de macro como uma expansão parametrizada, ou seja, uma expansão que depende dos parâmetros que recebe. Por exemplo: do código a seguir, será compilada a instrução *printf("%f\n", (b\*b - 4\*a\*c))* e não a instrução *printf("%f\n", DELTA(a, b, c))*. Além do mais, o compilador não saberia como compilar esta última instrução, já que não definimos nenhuma função com o identificador *DELTA*. Vejam:

```
#define DELTA(A, B, C) (B*B - 4*A*C)
...
float a = 1, b = 0, c = -1;
printf("%f\n", DELTA(a, b, c));
...
```

Por ser, tradicionalmente, uma boa prática de programação, definimos macros utilizando caracteres maiúsculos, para que seja fácil identificá-los em um código-fonte. Isto é, ao visualizarmos identificadores em que estão presentes apenas caracteres maiúsculos, podemos nos assegurar de que este identificador é uma macro, um *#define*.

É possível combinar diretivas utilizando diretivas condicionais, exemplo:

```
#ifdef _WIN32
    #define CLEAR "cls"
#else
    #define CLEAR "clear"
#endif
```

As diretivas acima definem uma macro com um comando utilizado para limpar a tela, de acordo com o sistema operacional em que o código-fonte for compilado. Quando um compilador está sendo executado no Windows, a macro `_WIN32` fica definida no ambiente e pode ser utilizada em diretivas do pré-processador. O trecho acima exemplifica uma maneira de escrever código que compila e executa corretamente em diferentes sistemas operacionais.

Observação: No Windows, o comando de terminal “cls” de fato limpa todo o terminal. No Linux, o comando equivalente “clear” apenas coloca algumas quebras de linha e sobe a tela.

## 2) Dos cabeçalhos

Neste trabalho, iremos conhecer dois novos cabeçalhos da biblioteca padrão do C: *stdlib.h* e *time.h*.

### 2.1) Do cabeçalho *stdlib.h*

Este cabeçalho nos fornece, entre outras, as interfaces de duas das funções da biblioteca padrão do C de que iremos precisar, isto é, nos diz como utilizar duas funções importantes para este trabalho:

```
void srand ( unsigned int seed );  
int rand ( void );
```

Computacionalmente, é impossível gerar números de forma verdadeiramente aleatória. No entanto, a biblioteca padrão do C nos fornece estas duas funções que geram números de uma forma pseudo-aleatória. Mais informações sobre números pseudo-aleatórios no link [http://pt.wikipedia.org/wiki/Sequ%C3%Aancia\\_pseudoaleat%C3%B3ria](http://pt.wikipedia.org/wiki/Sequ%C3%Aancia_pseudoaleat%C3%B3ria).

No sistema, existe uma variável chamada *seed*, que armazena a semente do algoritmo de geração de números pseudo-aleatórios. Usamos a função *srand()* para atribuir um valor a esta variável do sistema. A função *rand()* é utilizada para gerar os números pseudo-aleatórios. Cada chamada desta função retorna um número da sequência.

Fatos a se considerar:

- Toda vez que atribuímos um valor à semente, a sequência de números que a função *rand()* nos retorna é reiniciada;
- Para cada valor da semente, existe apenas **UMA** sequência de números pseudo-aleatórios;
- O valor padrão da semente, ou seja, o valor que a semente armazena no início de um programa, antes de qualquer chamada à função *srand()*, é **1**;
- O valor retornado pela função *rand()* sempre estará no intervalo  $[0, \text{RAND\_MAX}]$ , em que `RAND_MAX` é uma macro definida no cabeçalho *stdlib.h*.

Tendo em vista estes quatro fatos, é fácil gerar números pseudo-aleatórios em um dado intervalo. Por exemplo, para gerar um número no intervalo  $[0, 10]$ , basta utilizarmos a expressão *rand()%11*. Para gerar um número no intervalo  $[1, 10]$ , basta utilizarmos a expressão *rand()%10 + 1*.

Ao “juntarmos tudo”, veremos onde estas funções serão utilizadas.

### 2.2) Do cabeçalho *time.h*

Este cabeçalho nos fornece, entre outras coisas, uma função que iremos utilizar. A função *time()* retorna a quantidade de tempo, **sempre em segundos**, que já se passou desde *00:00 hours, Jan 1, 1970*

*UTC*. Atualmente, os valores retornados por esta função estão pouco acima de 1356048000. Além de retornar, é também opcional passar a referência de uma variável do tipo *time\_t*, onde a função possa armazenar este valor. Caso deseje que a função apenas retorne o valor, passe o valor 0 como argumento.

**3) switch** – Uma estrutura condicional extremamente recomendada para a implementação de **MENUS** é a estrutura *switch*. Esta estrutura é apenas uma maneira simplificada de escrever uma sequência *if-else if-else if-...-else if-else*, com a diferença de que o teste condicional realizado apenas verifica se um valor inteiro bate com uma das constantes selecionadas. Exemplo para o menu principal do seu jogo:

```
switch (menuMain()) {
case MENUMAIN_JOGAR: jogar(); break;
case MENUMAIN_CONFIGURACOES: configuracoes(); break;
case MENUMAIN_INSTRUcoes: instrucoes(); break;
case MENUMAIN_RANKING: ranking(); break;
case MENUMAIN_SAIR: sair(); break;
default:
exit(EXIT_FAILURE);
}
```

Uma sequência equivalente seria:

```
MenuMainOpt opt = menuMain();
if (opt == MENUMAIN_JOGAR)
jogar();
else if (opt == MENUMAIN_CONFIGURACOES)
configuracoes();
else if (opt == MENUMAIN_INSTRUcoes)
instrucoes();
else if (opt == MENUMAIN_RANKING)
ranking();
else if (opt == MENUMAIN_SAIR)
sair();
else
exit(EXIT_FAILURE);
```

#### 4) Das variáveis globais

No início do curso, vimos que variáveis globais podem confundir o programador ao manipular seus dados e decidimos não utilizá-las. No entanto, em alguns casos, a melhor opção pode ser utilizar variáveis globais, mas o cuidado deve ser mantido. O objetivo desta seção é mostrar formas de se detectar em quais situações uma variável deve ser global. Mais à frente, serão indicadas quais variáveis deverão ser globais neste trabalho.

Considere o programa a seguir, que lê um número *i* que indica o índice de um semestre de um aluno da UnB, um número *n* que indica a quantidade de disciplinas cursadas por este aluno neste semestre e uma sequência de *n* pares de números, um par para cada disciplina, em que o primeiro número indica quantos créditos a disciplina vale e o segundo número indica o valor do peso da menção

obtida pelo aluno. A seguir, o programa calcula a contribuição deste semestre para o IRA do aluno e mostra na tela este IRA calculado e o fatorial da porção inteira deste IRA.

```
#include <stdio.h>
#define MAX_DISCIPLINAS 10
void ler(int* i, int* n, int* credits, int* pesos_mencoes) {
    int j;
    printf("Entre com o indice do semestre e a qtde. de disciplinas cursadas: ");
    scanf("%d %d", i, n);
    getchar();
    for (j = 0; j < *n; ++j) {
        printf("Entre com a qtde. de credits que vale esta disciplina e o peso da
            mencao obtida pelo aluno: ");
        scanf("%d %d", &credits[j], &pesos_mencoes[j]);
        getchar();
    }
}

float calcularIraSemestre(int i, int n, int* credits, int* pesos_mencoes) {
    int j;
    float soma_sup = 0;
    float soma_inf = 0;
    /*regra da UnB que diz que o peso do semestre soh vai ate 6*/
    if (i > 6)
        i = 6;
    for (j = 0; j < n; ++j) {
        soma_sup += (i*credits[j]*pesos_mencoes[j]);
        soma_inf += (i*credits[j]);
    }
    return soma_sup/soma_inf;
}

int fat(int n) {
    int ret = 1;
    int j;
    if (n < 2)
        return 1;
    for (j = 1; j <= n; ++j)
        ret *= j;
    return ret;
}

int main() {
    float ira;
    int i, n, credits[MAX_DISCIPLINAS], pesos_mencoes[MAX_DISCIPLINAS];
    ler(&i, &n, credits, pesos_mencoes);
    ira = calcularIraSemestre(i, n, credits, pesos_mencoes);
    printf("IRA do semestre = %.4f\n", ira);
    printf("Fatorial do IRA castado para inteiro = %d\n", fat((int)ira));
    return 0;
}
```

Vamos observar dois fatos importantes.

**Primeiro fato:** os valores contidos nos parâmetros *credits* e *pesos\_mencoes* das funções *ler()* e *calcularIraSemestre()* serão sempre os mesmos, por mais que modifiquemos a função *main()* de várias formas. Por exemplo, criando um *loop* para ler vários semestres e outro para ler vários alunos, os vetores ainda estariam alocados nos mesmos lugares da memória e em todas as inúmeras chamadas das duas funções os parâmetros iriam sempre conter os mesmos endereços. Observemos também que isto **não** ocorre com os parâmetros *i* e *n* em **ambas** as funções, mas ocorre na função *ler()*. Isto é, a função *ler()* sempre deve conhecer o **endereço** das variáveis *i* e *n* declaradas na função *main()*, mas a função *calcularIraSemestre()* deve conhecer apenas o **conteúdo** destas variáveis; o que quer dizer que, nesta função, os valores destes parâmetros poderão ser diferentes a cada chamada que é feita.

**Segundo fato:** quantos programas de computador que fizemos poderiam vir a reutilizar as funções *ler()* e *calcularIraSemestre()*? Isto é, quantos programas podem querer ler um semestre no formato de um semestre da UnB e em seguida calcular o Índice de Rendimento Acadêmico estipulado pela UnB? Provavelmente, poucos. Poucos em relação ao número de programas de computador que poderiam vir a reutilizar, por exemplo, a função *fat()*, que, por acaso, é uma função que se encaixa bem no conceito de **relação funcional**: para cada entrada existe apenas uma única saída.

Suponha, agora, que as variáveis *i*, *n*, *créditos* e *pesos\_mencoes* tivessem sido declaradas no escopo global, ou seja, tivessem sido declaradas no topo do arquivo de código-fonte, de forma que todas as funções do programa as enxergassem. Tendo em vista os dois fatos observados e esta suposição, podemos chegar às seguintes conclusões:

- Poderíamos alterar a interface da função *ler()* para que ela não recebesse nenhum parâmetro, já que as variáveis sobre as quais ela deve operar estão no escopo global;
- Como discutimos no primeiro fato observado, as variáveis *i* e *n* são apenas lidas na função *calcularIraSemestre()* e por isso as passamos por cópia. No entanto, não haveria problema em, por exemplo, passar por referência, já que, como discutimos no segundo fato observado, não estamos desenvolvendo as funções deste programa com a intenção de reutilizá-las posteriormente em outro programa, ou seja, “o código é só nosso”. Neste caso, esbarramos novamente na conclusão logo acima, mas para o caso da função *calcularIraSemestre()*. Podemos alterar a interface desta função para que ela não receba nenhum parâmetro, já que as variáveis sobre as quais ela deve operar estão no escopo global;
- Estas alterações nas interfaces não afetariam o grau de reuso destas funções, já que, como discutimos no segundo fato observado, nem estamos preocupados com isto. Por outro lado, ainda obteríamos interfaces mais limpas, o que é sempre bom e desejado em qualquer situação, pois facilita tanto o uso da função, quanto o entendimento sobre o que ela faz.

Assim, vimos uma situação em que é possível, e até aconselhável, se utilizar variáveis globais. No entanto, este não é o único caso, mas assim mesmo não é possível criar uma regra geral para decidir se uma variável deve ser global, ou não. Cada caso deve ser avaliado isoladamente para que a decisão seja tomada da forma correta. Apesar de não existir uma regra clara, é possível avaliar cada caso tendo em mente o caráter da variável em questão. Variáveis de iteração, de troca de valores, de *flag* e de outros usos mais efêmeros, geralmente são locais. Variáveis que armazenam valores que precisam ser mantidos, como vetores, matrizes e outras estruturas mais complexas, correm riscos de serem globais, mas não é sempre assim.

No fim de tudo, o melhor é sempre partir do ponto de vista em que decidimos evitar variáveis globais, não o contrário. Isto é, não se deve procurar variáveis globais onde provavelmente não irá existir. Elas irão pular nos nossos caminhos por conta própria.

## 5) Juntando tudo

### Macros que devemos definir:

```
/*valor médio que a bomba levará para explodir (turnos)*/
#define MEDIA_TEMPO 6
/* erro da medida de tempo que a bomba levará para explodir (turnos)*/
#define ERRO_TEMPO 4
/* expressão que gera um número aleatório no intervalo [M - E, M + E]*/
#define RAND () (MEDIA_TEMPO + (rand()%(2*ERRO_TEMPO + 1) - ERRO_TEMPO))
```



Será necessário declarar nove variáveis globais no programa:

```
char tabuleiro[40][40];
int altura, largura;
int quantO, quantB, quantX;
int tamanhoB, loucuraX, tamanhoQ;
```

- **tabuleiro** – tabuleiro do jogo que será mostrado na tela e atualizado a cada turno.
- As seguintes variáveis poderão ser alteradas futuramente no menu de configurações:
  - **altura, largura** – dimensões do tabuleiro.
  - **quantO, quantB, quantX** – quantidade de cada tipo de inimigo que o jogo conterá em uma partida.
  - **tamanhoB, loucuraX, tamanhoQ** – variáveis que representam características de cada inimigo:
    - **tamanhoB** – armazena o maior tamanho do rastro que o inimigo ‘B’ pode gerar.
    - **loucuraX** – representa o intervalo de tempo, em turnos, em que o ‘X’ se movimenta aleatoriamente.
    - **tamanhoQ** – representa a área de explosão da bomba ‘Q’.

É possível que outras variáveis também precisem ser globais neste programa. Fica a cargo do programador tomar as decisões sobre quais variáveis devem ser globais. **Na correção do trabalho, estas decisões serão avaliadas.**

Algumas chamadas deverão ser feitas:

- Na inicialização do programa, faça a chamada *srand(time(0))*; . Esta chamada deve ser feita para que o algoritmo de geração de números pseudo-aleatórios possua uma semente diferente a cada nova execução do programa;
- Para determinar o tempo que uma bomba demora para explodir, use a expressão *RAND()*.
- **Função system()** - Uma sugestão interessante para o trabalho é utilizar a função *system()* do cabeçalho “*stdlib.h*” para limpar o terminal, a cada vez que o programa for exibir uma nova tela. Utilizando a dica do item 1 do Apêndice, basta realizar a chamada *system(CLEAR)* ;

Crie variáveis e nomes de funções mnemônicos!!!

Utilize variáveis globais com cautela conforme explicado acima.

Nas funções, utilize **passagem de parâmetros por valor e por referência**, conforme necessário.

Faça diversos testes no seu jogo, jogando é claro. Bom trabalho e boa diversão!

### **Observações Gerais:**

1. Incluir cabeçalho como comentário (ou seja, entre /\* \*/), no programa fonte, de acordo com os **critérios de avaliação dos trabalhos (Disponível no Moodle)**.
2. A data de entrega do programa é: **15/11/2017 (4a-feira) até às 23:55 hs.**