Transcrição da parte 3 do curso de programação em c

O primeiro passo para a implementação do Foge-Foge é pensar no mapa que nosso herói andará. Nele, precisamos representar tudo o que pode existir por lá, como paredes, monstros, o próprio herói, e lugares pela qual ele pode passar.

Podemos representar um mapa por meio de várias linhas e colunas. E, em cada posição, teríamos um caractere que representaria o que tem ali. Veja o exemplo do mapa abaixo, onde o ponto (.) significa "caminho livre", o arroba (@) representa nosso herói e os caracteres **|** e **-** representam paredes:

0123456789

0 |--------|

1 |...|..-.|

2 |..-|.@..|

3 |......-.|

4 |--------|

Nosso herói que está na linha 2, coluna 6, pode andar para esquerda, direita, cima ou baixo. Mas não pode andar duas vezes seguidas para a esquerda, por exemplo, pois há uma parede ali.

Podemos representar cada linha desse mapa por meio de um array de caracteres. Como cada linha tem 10 posições, um array de 10 posições resolveria. E como temos 5 linhas, precisamos de 5 desses arrays para representar todo o mapa:

char linha1[10];

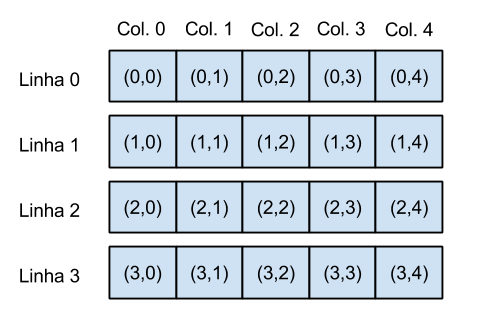
char linha2[10];

char linha3[10];

char linha4[10];

char linha5[10];

Claramente essa não é uma boa solução. Imagine se tivéssemos um mapa com 50 linhas? Precisaríamos ter 50 variáveis dessas. E manipulá-las não seria fácil. A solução para isso é deixar de usar vetores e passar a usar **matrizes**. Matrizes, assim como na matemática, é uma estrutura retangular de dados, ou seja, nosso "linhas e colunas".



Se quisermos uma matriz de caracteres de 5x10, declaramos uma variável para tal. Para declararmos matrizes em C, basta repetirmos duas vezes o conjunto de [], um para linhas e outro para colunas, respectivamente:

char mapa[5][10];

Acessar e alterar valores das posições é análogo ao uso de vetores, com a diferença que agora passamos ambas as posições:

// armazenando na posicao 0x0

mapa[0][0] = '|';

// imprimindo a posição 2x3

printf("%c", mapa[2][3]);

Para que nosso jogo fique interessante, vamos lê-lo de um arquivo. Nosso programa deve ler linha a linha e salvá-las na matriz mapa. Vamos começar abrindo o arquivo, do jeito que já conhecemos:

FILE\* f;

f = fopen("mapa.txt", "r");

if(f == 0) {

printf("Erro na leitura do mapa");

exit(1);

}

Agora precisamos mudar a declaração do nosso mapa. Apesar do mapa acima ter 10 caracteres na linha, precisamos lembrar que quando usarmos o fscanf(), passando uma string como máscara, ele colocará um \n ao final. Podemos também usar as funções de leitura que vimos no final do capítulo de I/O, como a fread(), e ler byte a byte, mas isso nos daria um trabalho desnecessário. Vamos declarar o array com uma posição a mais em cada linha, apenas para guardar o enter:

char mapa[5][10+1];

Com isso em mãos, podemos agora fazer o loop para capturar as linhas do arquivo. Apontamos mapa[i] como o lugar onde os caracteres devem ser colocados:

for(int i = 0; i < 5; i++) {

fscanf(f, "%s", mapa[i]);

}

Podemos imprimir o mapa, de maneira simples por enquanto:

for(int i = 0; i < 5; i++) {

printf("%s\n", mapa[i]);

}

Por fim, precisamos fechar o arquivo:

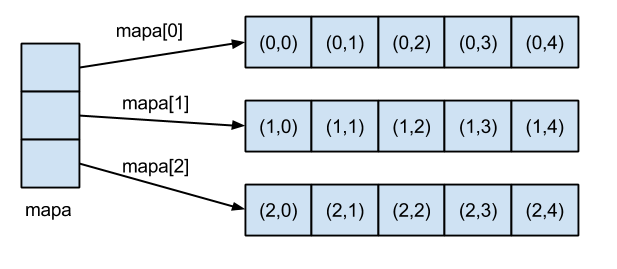
fclose(f);

Pronto. Já estamos lendo a matriz de mapa de um arquivo. Mas podemos melhorar ainda mais.

No curso anterior, discutimos que um array/vetor é simplesmente um ponteiro que aponta para uma posição da memória que tem espaços vazios para guardar as N posições declaradas.

Uma matriz é uma abstração ainda maior. Ela é um ponteiro que aponta para uma lista de arrays. Como arrays são ponteiros, então uma matriz é um ponteiro que aponta para uma lista de ponteiros.

Ou seja, mapa é um ponteiro que aponta para outros ponteiros. Isso significa que mapa[0] é um ponteiro que aponta para um array, mapa[1] é um ponteiro que aponta para outro array, e assim por diante. Veja na imagem abaixo:



**Ponteiros de ponteiros**

Em código, nada nos impede de declararmos uma matriz e depois declarar ponteiros que apontam para cada um dos vetores dentro dessa matriz. Veja o código abaixo. linha0[0] é igual a numeros[0][0], afinal tanto linha0 quanto numeros[0] apontam para o mesmo endereço de memória.

int numeros[5][10];

int\* linha0 = numeros[0];

int\* linha1 = numeros[1];

int\* linha2 = numeros[2];

int\* linha3 = numeros[3];

int\* linha4 = numeros[4];

// os numeros serao iguais, afinal

// ambos ponteiros apontam para o mesmo

// endereço de memória

printf("%d %d", linha0, numeros[0]);

Ou seja, nesse loop que faremos, guardaremos a primeira linha no array mapa[0], a segunda linha no array mapa[1] e assim por diante. Veja que no código acima, declaramos int\* para representar cada array que está dentro da matriz. Mas e se quiséssemos um novo ponteiro para apontar para a matriz como um todo? Como ela é um ponteiro que aponta para outros ponteiros de inteiro, a declaração é int\*\*, ou seja, duas estrelas para representar "ponteiro de ponteiro":

int numeros[5][10];

// o ponteiro copia é idêntico ao

// ponteiro numeros... ambos apontam

// para uma lista de ponteiros de inteiros

int\*\* copia = numeros;

// as duas operações abaixo são idênticas

numeros[0][0] = 10;

copia[0][0] = 10;

Ou seja, enquanto um array é um ponteiro para uma lista de posições de memória que estão livres para guardar o tipo escolhido, uma matriz é um ponteiro que aponta para uma lista de ponteiros, que por sua vez, apontam para posições de memória livres para guardar algo.

Mas e se quisermos mapas com tamanhos diferentes, para, por exemplo, diferenciar o nível do jogo? Quanto mais difícil, maior o mapa. O grande problema é: como variar o tamanho da matriz?

Uma possível estratégia seria ter uma matriz gigante, que suportasse o maior mapa possível. Por exemplo, uma matriz mapa[100][100]. Nosso programa então olharia apenas para a quantidade de linhas e colunas daquele mapa, em particular, que será sempre menor que 100x100.

Aqui, tomaremos outra decisão: a de gerar um array (ou matrizes) de tamanhos aleatórios. Leremos o arquivo, descobriremos o tamanho do mapa, e então, declararemos uma matriz com o tamanho ideal.

Para fazer isso, precisamos mudar a maneira de declararmos nosso array, alocando a memória necessária em tempo de execução. Para isso, usaremos a função malloc(). Ela recebe como parâmetro a quantidade de bytes que precisa ser alocado, e nos devolve um ponteiro para o primeiro byte alocado.

O código abaixo, por exemplo, aloca um único byte. Como ele nos devolve um ponteiro, o seu retorno pode ser armazenado em um ponteiro de char:

char\* letra = malloc(1);

\*letra = 'M';

Se quiséssemos guardar um int precisaríamos de mais de 1 byte, afinal inteiros são representados por 4 bytes. O código então seria algo como:

int\* numero = malloc(4);

\*numero = 20;

printf("%d", \*numero);

Mas como a quantidade de bytes que um inteiro ocupa pode variar de plataforma para plataforma, é melhor dizermos ao compilador para usar o tamanho do inteiro daquela plataforma. Para isso, usamos a instrução sizeof(int), que nos retorna o tamanho do inteiro correto:

int\* numero = malloc(sizeof(int));

\*numero = 20;

printf("%d", \*numero);

Como o malloc() simplesmente aloca a quantidade de bytes desejada e nos devolve um ponteiro, para declararmos um array de maneira dinâmica, basta apenas passarmos a quantidade de bytes corretos. Por exemplo, se quisermos um array de inteiros de 10 posições, precisamos passar 40 para ele (4 bytes por inteiro vezes 10 inteiros). Com esse número, sabemos que podemos manipular um ponteiro como se fosse um array:

// calcula o tamanho de bytes

int colunas = 10;

int memoria = sizeof(int) \* colunas;

// aloca memoria suficiente para o array

int\* numeros = malloc(memoria);

// usa o ponteiro como se fosse um array

numeros[2] = 10;

printf("%d", numeros[2]);

Já para uma matriz, temos mais de uma abordagem. Imagine uma matriz de 5 linhas por 10 colunas. Precisamos primeiro alocar um array para guardar 5 ponteiros de inteiros. Depois, alocar dez arrays de 5 arrays de 10 posições, e guardar cada um desses arrays nos ponteiros de inteiros declarados anteriormente.

Veja, em código:

int\*\* matriz;

int linhas = 5;

int colunas = 10;

// alocando espaço para as linhas,

// que guardam ponteiro de inteiro.

matriz = malloc(sizeof(int\*) \* linhas);

// agora, para cada linha, alocamos

// espaço para um array com 10 ("colunas") posições.

for(int i = 0; i < linhas; i++) {

matriz[i] = malloc(sizeof(int) \* colunas);

}

// agora podemos usar 'matriz' como uma matriz

matriz[2][3] = 10;

É isso que faremos então.

Vamos recomeçar nosso código já o separando em funções (código organizado sempre). Vamos declarar algumas variáveis globais, que serão importantes ao longo do jogo: o tamanho do mapa e o mapa em si:

#include <stdio.h>

#include <stdlib.h>

char\*\* mapa;

int linhas;

int colunas;

Nosso mapa.txt também terá um formato diferente. A primeira linha terá dois inteiros, indicando o tamanho do mapa que virá a seguir. No mapa atual, a primeira linha seria "5 10". E ler isso é fácil, dado que sabemos usar bem a função fscanf(). Usaremos esses números para então alocar o tamanho do mapa de maneira dinâmica, que será feito na função alocamapa():

void lemapa() {

FILE\* f;

f = fopen("mapa.txt", "r");

if(f == 0) {

printf("Erro na leitura do mapa");

exit(1);

}

fscanf(f, "%d %d", &linhas, &colunas);

alocamapa();

for(int i = 0; i < linhas; i++) {

fscanf(f, "%s", mapa[i]);

}

fclose(f);

}

A função alocamapa() contém um código bastante similar ao que usamos acima para entender alocação dinâmica. A diferença é que agora temos ponteiros para char. Apesar de sabermos que char é armazenado sempre em 1 bytes, vamos usar sizeof() para nunca termos problema. Lembre-se que precisamos adicionar mais 1 byte, para guardar o enter que é salvo pelo fscanf():

void alocamapa() {

mapa = malloc(sizeof(char\*) \* linhas);

for(int i = 0; i < linhas; i++) {

mapa[i] = malloc(sizeof(char) \* colunas + 1);

}

}

Nossa main por enquanto apenas lerá o mapa e o imprimirá, para termos a certeza que tudo está funcionando como gostaríamos:

int main() {

lemapa();

for(int i = 0; i < linhas; i++) {

printf("%s\n", mapa[i]);

}

}

Para também não termos problemas com ordem da declaração das funções, vamos desde já colocar todas as assinaturas dentro do fogefoge.h, que é importado no começo do nosso código-fonte:

// fogefoge.h

void alocamapa();

void lemapa();

Pronto. Nosso jogo agora é capaz de ler mapas de qualquer tamanho. Mas ainda temos um problema. Você se lembra que no capítulo de I/O, discutimos que todo recurso aberto deve ser fechado. Ou seja, se abrimos um arquivo, precisamos fechá-lo.

O mesmo acontece com memória que é alocada dinamicamente. Sempre que usamos malloc(), o sistema operacional nos reserva um pedaço da memória e não permite com que nenhum outro programa encoste nele. No entanto, nosso programa é o responsável também por liberar essa memória de volta ao SO. Fazemos isso por meio da função free(). Ou seja, para cada malloc(), precisamos de um free() ao final.

Veja a função liberamapa(), que navega pela matriz, liberando cada um dos arrays alocados dinamicamente. Ao final, libera também a matriz como um todo:

void liberamapa() {

for(int i = 0; i < linhas; i++) {

free(mapa[i]);

}

free(mapa);

}

Basta agora a invocarmos ao final do nosso programa, quando temos a certeza de que o mapa não será mais utilizado:

int main() {

lemapa();

for(int i = 0; i < linhas; i++) {

printf("%s\n", mapa[i]);

}

liberamapa();

}

Agora, ao rodarmos o programa, temos o mapa como saída:

|--------|

|...|..-.|

|..-|.@..|

|......-.|

|--------|

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/83a9b17ad588b37d9431).

Nesta aula, aprendemos:

* A declarar e manipular matrizes
* O que são ponteiros de ponteiros
* Que matrizes são, no fim, ponteiros de ponteiros
* A declarar memória dinamicamente, usando malloc()
* Que a instrução sizeof() nos devolve o tamanho em bytes de um tipo específico naquela plataforma
* Que devemos liberar a memória alocada dinamicamente ao fim, com free()

Nosso próximo passo é fazer nosso herói andar. Como todo jogo, usaremos a combinação de teclas A, S, D, W. A ideia será a seguinte: exibiremos o mapa, leremos uma entrada do teclado, faremos as atualizações necessárias no mapa, e repetiremos tudo de novo, até que o jogador ganhe ou perca.

Vamos começar com nosso loop principal, que será parecido com o do nosso jogo de forca. Usaremos a função scanf(), do jeito que você já está acostumado. Lembre-se do espaço em branco para que o enter seja ignorado:

int main() {

lemapa();

do {

imprimemapa();

char comando;

scanf(" %c", &comando);

move(comando);

} while (!acabou());

liberamapa();

}

A função acabou(), como o seu próprio nome diz, nos informará se o jogo acabou ou não. Por enquanto, vamos fazer uma função "falsa", que sempre nos retorna "0". Ou seja, por enquanto, o jogo nunca acaba:

int acabou() {

return 0;

}

A função imprimemapa() é também bastante simples. Ela apenas imprime o mapa, do jeito que ele está na matriz:

void imprimemapa() {

for(int i = 0; i < linhas; i++) {

printf("%s\n", mapa[i]);

}

}

Com o comando em mãos, podemos mover o herói para a direção que o jogador pediu. O primeiro passo do algoritmo deve ser localizar o herói. Para isso, não temos como fugir de varrer toda a matriz, usando dois loops encadeados:

void move(char direcao) {

int x;

int y;

for(int i = 0; i < linhas; i++) {

for(int j = 0; j < colunas; j++) {

if(mapa[i][j] == '@') {

x = i;

y = j;

break;

}

}

}

}

Em seguida, modifica o mapa de acordo com a direção escolhida: A vai para a esquerda, D para a direita, S para baixo e W para cima. Se o jogador andou pra esquerda, devemos deslocar o herói para lá, ou seja, para a linha, coluna-1). Se ele andou para direita, o herói vai para linha, coluna+1. Se andou para cima, vai para linha-1, coluna, e se andou para baixo, vai para linha+1, coluna. Ao final, devemos colocar como vazio a posição que o herói estava antes. Resolvemos isso com um switch:

void move(char direcao) {

// os fors aqui...

switch(direcao) {

case 'a':

mapa[x][y-1] = '@';

break;

case 'w':

mapa[x-1][y] = '@';

break;

case 's':

mapa[x+1][y] = '@';

break;

case 'd':

mapa[x][y+1] = '@';

break;

}

mapa[x][y] = '.';

}

Se você rodar o jogo, verá que ele já começa a se parecer com o que queremos. Conseguimos movimentar nosso personagem para todas as direções. Mas ainda estamos longe de terminá-lo: se digitamos uma letra inválida, o herói desaparece. Ele também atravessa paredes. Temos muito a corrigir.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/fbe62eb864d3c37516ae).

Repare que sempre que usamos o mapa, usamos também as variáveis linhas e colunas. Afinal, sempre precisamos delas para navegar na matriz. Como essas variáveis são globais, não temos problemas para usá-las. Mas e se precisássemos passá-las como parâmetros para alguma função? Seria trabalhoso; teríamos muitos parâmetros.

Pela primeira vez temos um conjunto de variáveis que "não fazem sentido separadas". A matriz só faz sentido se tiver uma linha e coluna junto. Precisamos agrupá-las, e garantir que elas sempre serão "transportadas" juntas. Para tal, usaremos uma **struct**. Estruturas (ou, em C, struct) são uma maneira que temos para agrupar variáveis. Ou seja, definiremos que a estrutura "mapa" contém uma matriz (o ponteiro de ponteiro de char), uma variável para guardar a quantidade de linhas (int) e outra para guardar a quantidade de colunas (também int). Com essa estrutura definida, podemos declarar variáveis que a seguem. Dessa forma, garantiremos sempre que mapas tem uma matriz, e a quantidade de linhas e colunas.

Veja a declaração da struct abaixo, que colocamos dentro do nosso fogefoge.h. Dentro dele, declaramos a matriz e os dois inteiros para guardar as suas dimensões:

struct mapa {

char\*\* matriz;

int linhas;

int colunas;

};

Como dissemos anteriormente, com essa struct em mãos, podemos fazer uso dela e declarar variáveis desse tipo. Mas diferente de um inteiro ou um char que usamos até então, ela tem variáveis dentro dela. Para acessá-las, usamos ponto (.). Veja o trecho de código abaixo, por exemplo, onde definimos um mapa e setamos os valores para linhas e colunas. Repare na palavra struct no começo da declaração:

struct mapa facil;

char tabuleiro[10][20];

facil.matriz = &tabuleiro;

facil.linhas = 10;

facil.colunas = 20;

printf("O mapa tem tamanho %d x %d", facil.linhas, facil.colunas);

Vamos fazer as devidas alterações no código, usando agora a struct que definimos. A começar pelas variáveis globais, que são substituídas:

struct mapa m;

Em todo lugar que fazia uso de linhas ou colunas, precisamos trocar para m.linhas e m.colunas. Em todo lugar que tínhamos a matriz mapa, agora precisamos usar m.matriz.

Um detalhe importante é perceber precedência de operadores. Por exemplo, dentro da função lemapa(), temos o fscanf(), que agora ficou parecido com o código abaixo:

fscanf(f, "%d %d", &m.linhas, &m.colunas);

A instrução &m.linhas é processada corretamente. Passamos o endereço da variável linhas que está dentro da struct m. Uma outra interpretação dessa instrução seria que o & é para a variável m, ou seja, gostaríamos do endereço de memória da estrutura m como um todo. Para evitar essa possível ambiguidade, é comum fazermos uso de parênteses para deixar claro a ordem de precedência:

fscanf(f, "%d %d", &(m.linhas), &(m.colunas));

A função move(), com todas as alterações, fica como a abaixo:

void move(char direcao) {

int x;

int y;

for(int i = 0; i < m.linhas; i++) {

for(int j = 0; j < m.colunas; j++) {

if(m.matriz[i][j] == '@') {

x = i;

y = j;

break;

}

}

}

switch(direcao) {

case 'a':

m.matriz[x][y-1] = '@';

break;

case 'w':

m.matriz[x-1][y] = '@';

break;

case 's':

m.matriz[x+1][y] = '@';

break;

case 'd':

m.matriz[x][y+1] = '@';

break;

}

m.matriz[x][y] = '.';

}

Apesar da sensação de que agora temos muito mais código escrito, usar structs para agrupar dados que devem ficar perto é uma boa prática. Você verá que, em breve, deixaremos de usar variáveis globais, e começaremos a passar essas structs pra lá e pra cá. Mas, como elas agrupam tudo o que precisamos, fica fácil: a função receberá apenas um ponteiro para a estrutura.

Repare que sempre que precisamos declarar uma variável do tipo da estrutura, precisamos usar a palavra struct antes. Por exemplo, struct mapa m. Podemos melhorar isso, fazendo uso da instrução typedef. Essa instrução nos possibilita dar um outro nome a algum tipo já existente.

Por exemplo, vamos apelidar struct mapa de MAPA, dentro do nosso header file:

typedef struct mapa MAPA;

Graças ao typedef, agora podemos declarar variáveis do tipo MAPA sem a palavra struct. Veja:

MAPA m;

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/970a321e2cf46ca658fc).

Você reparou também que a maioria das funções que escrevemos até então tem a palavra "mapa" em alguma parte do nome? A estrutura mapa, bem como as funções alocamapa(), lemapa(), liberamapa() e imprimemapa() fazem algo relacionado ao mapa. As funções restantes, acabou(), move() e main() apenas fazem uso da estrutura e/ou funções do mapa.

Podemos pensar então em isolar todas as funções em um arquivo específico, por exemplo, mapa.c. Mas o que ganhamos com isso? Reuso e manutenibilidade. Ter um arquivo dedicado somente a lidar com mapas nos possibilita utilizá-lo em outras aplicações (mapas podem ser úteis em muitos outros jogos). Além disso, sabemos que se algo der errado com mapas, muito provavelmente esse código está dentro do arquivo específico.

A discussão aqui é similar a que fizemos quando chegamos à conclusão de que funções eram úteis para isolar e reutilizar código. Aqui, a mesma coisa, mas em um nível maior. Teremos um arquivo que agrupará um conjunto de funções que tem uma relação entre si.

Vamos criar o arquivo mapa.h, com a definição da struct e as assinaturas das funções que lidam com mapas:

struct mapa {

char\*\* matriz;

int linhas;

int colunas;

};

typedef struct mapa MAPA;

void alocamapa();

void lemapa();

void liberamapa();

void imprimemapa();

Já o arquivo mapa.c contém a implementação dessas funções:

// mapa.c

#include <stdio.h>

#include <stdlib.h>

#include "mapa.h"

void lemapa() {

// codigo aqui

}

void alocamapa() {

// codigo aqui

}

void liberamapa() {

// codigo aqui

}

void imprimemapa() {

// codigo aqui

}

Já no fogefoge.c, incluímos o header dos mapas. Afinal, precisamos dessas funções disponíveis também nesse arquivo (e, obviamente, retiramos as funções de mapa, que agora estão em seu arquivo particular):

#include <stdio.h>

#include <stdlib.h>

#include "fogefoge.h"

#include "mapa.h"

Vamos compilar nosso programa. A linha de compilação agora precisa contemplar ambos os arquivos. Para isso, basta passar a lista deles para o gcc:

gcc fogefoge.c mapa.c -o fogefoge.out

Entretanto, o compilador nesse momento nos devolve muitos erros. A maioria deles reclamando sobre a não existência da variável m no arquivo mapa.c. E faz sentido: a variável m está declarada em outro arquivo.

Variáveis, mesmo que globais, são visíveis apenas no arquivo em que foram declaradas. Ou seja, a variável m é visível apenas dentro do fogefoge.c. E faz sentido ela ser declarada lá, afinal ela representa um mapa do nosso jogo. Não é algo genérico para colocarmos dentro do mapa.c.

Ainda assim queremos separar as funções de mapa do resto. Portanto, não teremos outra solução a não ser fazer as funções que lidam com mapa, receber um mapa como parâmetro. Vamos pegar, por exemplo, a função alocamapa() e fazê-la receber um mapa como parâmetro:

void alocamapa(mapa m) {

m.matriz = malloc(sizeof(char\*) \* m.linhas);

for(int i = 0; i < m.linhas; i++) {

m.matriz[i] = malloc(sizeof(char) \* m.colunas + 1);

}

}

O código agora compilará. Mas, da forma com que ele está escrito, teremos um problema. Para entendê-lo, vamos criar um outro programa bastante simples, que ilustrará o que precisamos.

O código abaixo declara uma simples estrutura quadrado com dois inteiros. A função dobra() dobra os valores de x e y. A função main() imprime os valores dos quadrados antes e depois deles serem dobrados.

#include <stdio.h>

struct quadrado {

int x;

int y;

};

typedef struct quadrado QUADRADO;

void dobra(QUADRADO q) {

q.x = q.x \* 2;

q.y = q.y \* 2;

}

int main() {

QUADRADO q1;

q1.x = 20;

q1.y = 35;

printf("%d %d\n", q1.x, q1.y);

dobra(q1);

printf("%d %d\n", q1.x, q1.y);

}

Mas, ao rodarmos o programa, temos uma saída que não esperamos: os resultados são iguais, antes e depois da função de dobra:

20 35

20 35

A razão do resultado é fácil de ser entendida. E você, na verdade, já sabe, mas pode ter-se confundindo por causa da struct. Veja o código abaixo, mais simples ainda, que, ao invés de dobrar o valor dentro de uma struct, dobra o valor de uma variável n:

void dobra(int n) {

n = n \* 2;

}

int main() {

int n = 10;

printf("%d", n);

dobra(n);

printf("%d", n);

}

A saída do programa é 10 10. Veja que a variável n do método main é diferente da variável n do método dobra. A máquina, no momento em que invocamos o método dobra() gera uma cópia da variável n que está em main e a passa para a variável n da função dobra(). Ou seja, passagem por cópia. O conteúdo da variável é **copiado** para o parâmetro. Já vimos isso durante nosso capítulo de ponteiros. Foi exatamente por isso que optamos por usar ponteiros, aliás.

O mesmo aconteceu com a struct. Ao passarmos a variável q1 declarada na main para a função dobra, a máquina passou apenas uma cópia da struct original. As mudanças, portanto, foram feitas na cópia. Se quisermos modificar a struct original (da mesma forma que se quisermos modificar qualquer outra variável original), precisamos lidar com ponteiros.

A função alocamapa() deve receber um ponteiro de mapa, para que suas alterações sejam feitas sempre na posição de memória da variável original. Ou seja, passagem por referência:

void alocamapa(mapa\* m) {

// ...

}

Agora precisamos mudar o corpo da função. Afinal, antes fazíamos m.linhas para acessar o conteúdo da variável linhas, dentro de m. Mas agora, m é um ponteiro para um mapa. Ou seja, precisamos primeiro ir para o conteúdo do ponteiro, para depois usar a variável linhas. Em código, (\*m).linhas. Veja a primeira linha dessa função, modificada:

(\*m).matriz = malloc(sizeof(char\*) \* (\*m).linhas);

Dado que isso é uma operação bastante comum (acessar primeiro o conteúdo do ponteiro e depois uma variável qualquer dentro da struct), a linguagem C possui um açúcar sintático que nos ajuda a escrever a mesma coisa de maneira mais simples. Para isso, usamos uma seta (->). A seta faz a mesma coisa que o "estrela-ponto" que usamos no código anterior. Por exemplo, m->linhas. Veja a primeira linha:

m->matriz = malloc(sizeof(char\*) \* m->linhas);

Veja a função alocamapa() inteira:

void alocamapa(MAPA\* m) {

m->matriz = malloc(sizeof(char\*) \* m->linhas);

for(int i = 0; i < m->linhas; i++) {

m->matriz[i] = malloc(sizeof(char) \* m->colunas + 1);

}

}

Todas as outras funções do arquivo mapa.c devem seguir a mesma abordagem: receber um ponteiro de mapa como parâmetro, e fazer uso da seta para manipular esse mapa. As funções agora têm essa assinatura:

struct mapa {

char\*\* matriz;

int linhas;

int colunas;

};

typedef struct mapa MAPA;

void alocamapa(MAPA\* m);

void lemapa(MAPA\* m);

void liberamapa(MAPA\* m);

void imprimemapa(MAPA\* m);

### A ordem importa?

No header file, você precisa colocar o struct mapa antes das declarações das funções. Afinal, na assinatura da função, usamos o tipo MAPA, e ele precisa ser declarado antes. Veja o código acima com atenção.

Não podemos esquecer também de passar o parâmetro para a função alocamapa(), invocada dentro da lemapa():

void lemapa(MAPA\* m) {

// ...

fscanf(f, "%d %d", &(m->linhas), &(m->colunas));

alocamapa(m);

// ...

}

De volta ao fogefoge.c, precisamos agora passar o endereço de memória da variável m para as funções (usando &). Nesse momento, as alterações concentram-se na função main:

int main() {

lemapa(&m);

do {

imprimemapa(&m);

char comando;

scanf(" %c", &comando);

move(comando);

} while (!acabou());

liberamapa(&m);

}

Nosso jogo agora voltou a compilar e a funcionar. Repare que o comportamento é o mesmo que já tínhamos antes de começar toda essa refatoração. A diferença é que agora temos nosso código ainda mais fácil de ser lido e mantido.

Variáveis declaradas como globais são globais para todo o programa. Mas, para que um arquivo enxergue uma variável global definida em outro arquivo, precisamos fazer uso da palavra extern.

Imagine, por exemplo, uma variável int n; for definida no arquivo a.c. Para usarmos a mesma variável no arquivo b.c, precisaríamos redeclara-la com extern int n;. Agora sim, a variável n, global, é visível por ambos arquivos. Repare que só um arquivo a realmente declara, e os outros deixam claro que ela é "externa".

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/effe539d9ad12d2312e5).

Analise agora a função move(). Analisar funções é algo comum em nosso dia a dia de trabalho, afinal, se uma delas não funcionar bem, ou levar muito tempo para executar, podemos ter problemas em nosso software.

Quando analisamos uma função, geralmente vemos o trabalho que ela tem em relação aos dados de entrada que ela recebe. Nesse caso, consideramos o mapa como seu dado de entrada. Veja que sempre que invocamos a função, os dois fors garantem que toda a matriz é percorrida. Se o herói estiver na última posição inferior a direita, então o algoritmo passeará por **todas** as posições da matriz.

Ou seja, quanto maior o mapa, mais lerda essa função será. Se fôssemos definir isso em termos matemáticos, diríamos que essa função tem a complexidade de no máximo linhas \* colunas. Aqui não entraremos em detalhes sobre **análise de algoritmos**, que é uma disciplina fundamental em computação. No entanto, perceba desde já que se você tem algoritmos que passeiam por todo um array ou matriz, quanto maior esse array, mais tempo essa função levará. Será que conseguimos então "economizar" no número de vezes que varremos a matriz?

Repare que esses loops servem apenas para encontrar a posição atual do herói no mapa. Uma solução para isso então é termos duas variáveis globais, que guardariam o X e Y do herói. Dessa forma, não precisaríamos buscá-lo o tempo todo; já teríamos essa informação salva.

Vamos declarar duas variáveis globais para isso:

int x;

int y;

Com essas duas variáveis, podemos eliminar então os fors encadeados, e a função move() fica ainda menor. Nesse momento, repare que além de menor, ela também é mais rápida: agora ela não varre toda a matriz para fazer seu trabalho. Independente do tamanho da matriz, ela gastará sempre o mesmo tempo para executar. Dizemos que essa função leva tempo constante para executar:

void move(char direcao) {

switch(direcao) {

case 'a':

m.matriz[x][y-1] = '@';

break;

case 'w':

m.matriz[x-1][y] = '@';

break;

case 's':

m.matriz[x+1][y] = '@';

break;

case 'd':

m.matriz[x][y+1] = '@';

break;

}

m.matriz[x][y] = '.';

}

Precisamos agora encontrar o lugar ideal para encontrar a posição do herói pela primeira (e única) vez. Podemos fazer isso logo após a invocação da função lemapa() no método main(). Mas antes, vamos declarar outra estrutura. Perceba que sempre que há um X, há também um Y. Por quê não ter a estrutura posicao. Como uma posição é relacionado a um mapa, essa estrutura deve ser declarado no header file de mapa:

struct posicao {

int x;

int y;

};

typedef struct posicao POSICAO;

Vamos agora criar a função que localiza o herói e guarda sua posição numa estrutura dessas. Repare que recebemos o ponteiro de ambas estruturas. Repare também nas setas, usadas para armazenar os valores dentro da posição de memória que os ponteiros apontam:

void encontramapa(MAPA\* m, POSICAO\* p, char c) {

for(int i = 0; i < m->linhas; i++) {

for(int j = 0; j < m->colunas; j++) {

if(m->matriz[i][j] == c) {

p->x = i;

p->y = j;

break;

}

}

}

}

Precisamos agora declarar uma POSICAO que guardará a posição do nosso herói, e invocar a função encontramapa() logo após lermos o mapa:

// variaveis globais

MAPA m;

POSICAO heroi;

// ...

int main() {

lemapa(&m);

encontramapa(&m, &heroi, '@');

// ...

}

Por fim, precisamos fazer com que a função move() faça uso agora dos X e Y que estão dentro do heroi. Além disso, precisamos também atualizar essa variável sempre que o herói andar. O faremos dentro de cada opção do switch. Repare também que fazemos a substituição da posição atual do herói por vazio no começo da função; afinal, já que alteramos os valores da variável heroi, após o switch a variável já aponta para a nova posição do herói.

void move(char direcao) {

m.matriz[heroi.x][heroi.y] = '.';

switch(direcao) {

case 'a':

m.matriz[heroi.x][heroi.y-1] = '@';

heroi.y--;

break;

case 'w':

m.matriz[heroi.x-1][heroi.y] = '@';

heroi.x--;

break;

case 's':

m.matriz[heroi.x+1][heroi.y] = '@';

heroi.x++;

break;

case 'd':

m.matriz[heroi.x][heroi.y+1] = '@';

heroi.y++;

break;

}

}

Nosso jogo continua funcionando. Agora com um algoritmo ainda mais performático. Isso é, em alto nível, o que chamamos de análise de algoritmo. É olhar para ele e avaliar seu desempenho. Esse desempenho pode ser avaliado de diferentes maneiras: consumo de memória, tempo de processamento, espaço em disco, e etc.

Geralmente estamos preocupados com o tempo de processamento. Antes, nossa função move() levava um tempo proporcional ao tamanho da matriz. Ou seja, quanto maior a matriz, mais tempo ele levava. Vimos ainda que no pior caso, o tempo do algoritmo poderia ser proporcional ao número de linhas vezes o número de colunas (**linhas x colunas**), justamente porque ele era obrigado a passear por toda a matriz. Já a nova versão, o tempo de execução da função não varia de acordo com o tamanho da matriz. Ou seja, o tempo dela é sempre o mesmo: é constante.

É isso que você aprenderá na disciplina de análise de algoritmos. Obviamente, por lá você aprenderá todo o ferramental matemático para avaliar e expressar o tempo de execução desses algoritmos. Não é o foco desse curso, claramente. Mas perceba desde já a importância do tema.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/3faed15c46f8466e88c3).

Nesta aula, aprendemos:

* Por que é importante deixar perto dados que se relacionam.
* A definir structs.
* A declarar variáveis do tipo da struct.
* A usar o typedef para definir um nome de tipo e vinculá-lo a uma struct.
* A separar funções em diferentes arquivos.
* A compilar mais de um arquivo .c junto, para formar um só programa.
* A passar structs como referência.
* A usar o operador -> para manipular ponteiros de structs.
* Que análise de algoritmos é uma importante área da computação.

Está na hora de continuarmos nosso jogo. Afinal, ele é, de longe, o mais complicado dos três. A primeira coisa a fazer é acertar o movimento do nosso herói. Nesse momento, se apertarmos alguma tecla que não é de movimentação, o herói some do mapa. Mais ainda, ele não pode andar por onde quiser: ele deve parar nas paredes e extremos do mapa.

Vamos começar ignorando qualquer comando que não seja de andar. Nesse momento do código, podemos fazer isso dentro da própria função move(). Um simples if resolve: se a letra não for "a, s, d, w", então sai da função sem fazer nada.

Apesar de até então termos usado a palavra return para devolver valores em uma função, podemos usá-la para terminar a execução de um método void. Nesses casos, a palavra return não vem acompanhada de nada. Repare que esse if é comprido e é invertido. Ou seja, se a letra não for nem "a", nem "w", nem "s", nem "d", então sai da função:

void move(char direcao) {

if(

direcao != 'a' &&

direcao != 'w' &&

direcao != 's' &&

direcao != 'd'

) return;

// ...

}

A próxima validação é tratar o caso da parede e dos extremos. Nosso herói só pode andar se a casa na posição escolhida é um "." (ou seja, vazio). E, claro, se ele já estiver na última casa a direita, ele não pode ir mais a direita; o mesmo para esquerda, cima e baixo.

Para fazer esse if, precisaremos mudar um pouco nossa implementação. O cálculo da próxima posição é feita dentro do switch. Ali mesmo calculamos a posição e já modificamos a matriz, colocando o herói. Agora precisamos validar antes de colocá-lo lá. Para isso, criaremos duas variáveis auxiliares que armazenarão a posição em que o herói deve ser colocado. E é nelas que faremos as validações.

void move(char direcao) {

if(

direcao != 'a' &&

direcao != 'w' &&

direcao != 's' &&

direcao != 'd'

) return;

int proximox = heroi.x;

int proximoy = heroi.y;

switch(direcao) {

case 'a':

proximoy--;

break;

case 'w':

proximox--;

break;

case 's':

proximox++;

break;

case 'd':

proximoy++;

break;

}

}

A validação é fácil. Sabemos que proximox deve ser menor do que a quantidade de linhas da matriz, e que proximoy deve ser menor do que a quantidade de colunas. Vamos escrever aqui o if ao contrário, pois usaremos a mesma estratégia com o caractere da direção: se proximox for maior ou igual a quantidade de linhas, então saímos da função. A mesma coisa para o proximoy:

if(proximox >= m.linhas)

return;

if(proximoy >= m.colunas)

return;

Repare que não abrimos chaves no if. Podemos fazer isso. Nesses casos, "a única coisa dentro do if" é a próxima instrução. Ou seja, o que fizemos acima é o mesmo que fazer o código abaixo:

if(proximox >= m.linhas) {

return;

}

if(proximoy >= m.colunas) {

return;

}

A diferença é que essa ocupa mais linhas. Isso é um pouco de gosto de cada desenvolvedor. Na maioria das vezes optamos por abrir chaves, afinal isso facilita a legibilidade. Mas, nesse caso vamos deixar de usá-las, afinal os ifs são muito parecidos e sequenciais. Nesse caso, a falta das chaves não atrapalha a legibilidade.

A próxima etapa é garantir que a nova posição é vazia. Só assim ele poderá ir para lá. Basta adicionar mais um if, também invertido, que verifica se a posição contém um "." (ponto):

if(m.matriz[proximox][proximoy] != '.')

return;

Isso quer dizer que se nosso código sobreviveu por todos esses ifs, então o herói está pronto para ir para essa posição. Precisamos agora fazer a troca: colocar o herói na posição nova, marcar a antiga posição dele como vazia, e acertar nossa variável heroi com as novas coordenadas:

m.matriz[proximox][proximoy] = '@';

m.matriz[heroi.x][heroi.y] = '.';

heroi.x = proximox;

heroi.y = proximoy;

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/18087848e7bd1b78041d).

### E se fizéssemos sem os ifs invertidos?

Nesse caso, teríamos uma quantidade grande de ifs aninhados. Veja que a função só deve executar se a tecla digitada é uma direção, se proximox for menor que a quantidade de linhas, se proximoy for menor que a quantidade de colunas e se a próxima posição é vazia.

Ou seja, teríamos muitos ifs aninhados, e o código ficaria mais ou menos como o abaixo. Veja que o código real, que importa ficaria "lá pra frente", com muitas indentações. Isso dificulta a legibilidade dele. Por isso que é comum fazermos o contrário: se uma condição falha saímos da função.

if(primeira regra está certa) {

if(segunda regra está certa) {

if(terceira regra está certa) {

// ufa, agora sim vou fazer o que eu quero

printf("Tudo é valido!");

}

}

}

Pronto. Nosso herói agora só anda por onde é realmente válido.

## Novamente, responsabilidades

Repare que apesar do andar do herói funcionar bem da forma que queríamos, nosso código não está perfeito. A função move() é muito grande, e tem responsabilidades demais. Quando dizemos responsabilidades, estamos nos referindo à quantidade de regras de negócio diferentes que está dentro dela. Veja que ela é "responsável" por saber se o caractere é uma direção válida, é responsável por calcular a próxima posição do herói, é responsável por saber se a nova posição está dentro do mapa e é válida, e é responsável por fazer as manipulações na matriz.

É simplesmente muita coisa para uma única função. Aliás, tem código aí que só tem relação com mapas, e já sabemos que tudo que diz respeito à mapas deveria estar no mapa.c. Esse tipo de situação é bastante comum no dia a dia. Afinal, estávamos muito preocupados em fazer a regra de negócio funcionar; acabamos não pensando na qualidade do código. É por isso que precisamos sempre refletir sobre o estado atual do código, e melhorá-lo, se necessário.

Vamos começar a quebrar essa função. A primeira coisa que podemos fazer é levar esse if que verifica se a letra digitada é uma direção, que é grande, para uma função própria. A função se chamará ehdirecao(char) e nos retornará um inteiro 0 ou 1. Repare que precisamos inverter todo o if, pois agora a função nos retornará verdadeiro se o caractere é uma direção:

int ehdirecao(char direcao) {

return

direcao == 'a' ||

direcao == 'w' ||

direcao == 's' ||

direcao == 'd';

}

A função move() então já começa a ficar menor e mais fácil de ser entendida. Repare que antes o desenvolvedor precisava ler o if e entendê-lo. Agora não: o nome da função deixa bem claro o que ela faz. O desenvolvedor não precisa nem ler a implementação se não quiser:

void move(char direcao) {

if(!ehdirecao(direcao))

return;

// ...

}

Podemos deixar o switch dentro da função. Afinal, calcular a posição é parte de "mover". Mas os dois próximos ifs podem ir para dentro das funções de mapa; eles nos dizem se uma determinada posição do mapa é válida ou não. Aqui, nossa estratégia será retornar 0, caso alguma das posições seja inválida ou, no fim, retornar 1, dizendo que x e y estão dentro dos limites do mapa. Em código:

int ehvalida(MAPA\* m, int x, int y) {

if(x >= m->linhas)

return 0;

if(y >= m->colunas)

return 0;

return 1;

}

O próximo if nos diz se a próxima posição do mapa é vazia. Isso também pode ir para dentro do mapa. Uma simples função que nos devolve um booleano deixará nosso código muito mais claro. Como ela manipula mapa, logo está dentro do mapa.c. Repare que invertemos a condição do if para que ela nos diga se aquela posição é ou não vazia:

int ehVazia(MAPA\* m, int x, int y) {

return m->matriz[x][y] == '.';

}

Os ifs então devem ser substituídos por invocações a essas funções. Veja que apesar de termos diminuído pouco a quantidade de condicionais, esse código é muito mais fácil de ser entendido. Afinal, a palavra ehvalida deixa mais claro o que acontece do que aquele if grande. Precisamos ler menos, o que é bom:

if(!ehvalida(&m, proximox, proximoy))

return;

if(!ehvazia(&m, proximox, proximoy))

return;

Vamos agora colocar as próximas duas linhas, que são responsáveis por mover o que está em uma posição, para a nova posição. Essa também é uma função que deve estar no mapa (e que provavelmente será utilizada mais a frente, pois teremos fantasmas no jogo, que também andarão pra lá e pra cá).

A função andanomapa() receberá 5 argumentos: um ponteiro de mapa, X e Y de origem, e X e Y de destino. Ela é bem simples: primeiro, ela armazena o personagem (ou qualquer coisa) que estava na posição antiga. Em seguida, ela faz a troca: coloca o personagem na posição de destino, e troca por vazio a posição antiga:

void andanomapa(MAPA\* m, int xorigem, int yorigem,

int xdestino, int ydestino) {

char personagem = m->matriz[xorigem][yorigem];

m->matriz[xdestino][ydestino] = personagem;

m->matriz[xorigem][yorigem] = '.';

}

O código também fica mais simples no move(), já que basta invocar agora a nossa nova função que anda um personagem no mapa, e trocar os valores da posição do nosso heroi:

andanomapa(&m, heroi.x, heroi.y, proximox, proximoy);

heroi.x = proximox;

heroi.y = proximoy;

Pronto. A função move() agora é muito mais coesa. Ou seja, ela faz muito menos tarefas, e delega para outras funções. Temos agora funções menores e mais fáceis de serem lidas, mantidas e reutilizadas. Isso é programar.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/43d5cf4b0489a7d839d4).

Desde o primeiro jogo, discutimos os problemas de termos números mágicos espalhados pela aplicação. No momento, não temos números mágicos, mas sim "chars mágicos". Veja que temos em nosso código, as letras 'a', 'd', 's', 'w', que significam as possíveis direções, ou mesmo '@', '.' e '|', que indicam os possíveis personagens no mapa.

Vamos criar constantes para isso. Você já conhece as vantagens: maior legibilidade, afinal uma constante tem um nome, e facilidade na manutenção, afinal basta alterar o valor da constante para a alteração propagar.

As constantes dizem respeito aos mapas, vão para o mapa.h:

#define HEROI '@'

#define VAZIO '.'

#define PAREDE\_VERTICAL '|'

#define PAREDE\_HORIZONTAL '-'

Já as constantes que dizem respeito ao jogo, vão para o fogefoge.h:

#define CIMA 'w'

#define BAIXO 's'

#define DIREITA 'd'

#define ESQUERDA 'a'

Agora, basta fazermos uso delas ao invés de termos esses caracteres fixos. A função ehdirecao(), por exemplo, fica ainda mais legível. Também faremos uso dessas constantes dentro do switch, na função move(). Repare que, se decidirmos mudar as teclas do jogo, bastará alterarmos as constantes.

int ehdirecao(char direcao) {

return

direcao == ESQUERDA ||

direcao == CIMA ||

direcao == BAIXO ||

direcao == DIREITA;

}

Generalize o que fizemos nesse vídeo. Perceba que no fundo, toda vez que falamos de boas práticas de código, fazemos a mesma coisa: isolar trechos de código para que possamos reutilizá-lo em diversos pontos da aplicação, de forma com que fique fácil saber o que ele faz apenas lendo o seu nome.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/9edc5bdd8edacfbc2072).

O próximo passo agora é fazer os fantasmas andarem pelo mapa. Ainda não definimos como representaremos um fantasma no mapa. Assim como nos outros casos, escolheremos uma caractere para tal: F.

Vamos colocar alguns fantasmas no mapa, para começarmos nossos testes:

5 10

|--------|

|.F.|....|

|..-|.@..|

|..F.....|

|--------|

Agora precisamos movimentá-los no mapa. O algoritmo pode ser o quão complicado quisermos: eles poderiam ter algum tipo de inteligência artificial, por exemplo. Mas começaremos com algo simples, como fazendo todos eles andarem, por enquanto, somente para a direita.

Para tal, varreremos a matriz em busca dos fantasmas. E, assim que os encontrarmos, andaremos uma posição à direita em cada um deles. A função fantasmas() será responsável por isso. Ao encontrarmos um fantasma, verificaremos se a próxima posição a direita é válida. Se sim, então o fantasma andará para a direita:

void fantasmas() {

for(int i = 0; i < m.linhas; i++) {

for(int j = 0; j < m.colunas; j++) {

if(m.matriz[i][j] == FANTASMA) {

if(ehvalida(&m, i, j+1)) {

andanomapa(&m, i, j, i, j+1);

}

}

}

}

}

Invocaremos a função fantasmas() logo após a função move(), dentro do nosso loop principal do jogo:

do {

imprimemapa(&m);

char comando;

scanf(" %c", &comando);

move(comando);

fantasmas();

} while (!acabou());

Se rodarmos nosso programa do jeito que está, teremos uma surpresa: o fantasma andará, não só uma, mas todas as casas que ele puder para a direita. Entender o porquê isso aconteceu é importante, pois nos dará um entendimento ainda maior sobre algoritmos e lógica de programação.

Veja, por exemplo, o fantasma que está em (3,3) do mapa. Nosso algoritmo o encontrará quando i for igual a 3, e j for também igual a 3. A próxima posição, (3,4) é vazia, e portanto movemos o fantasma. Até então, tudo funcionando de acordo.

Entretanto, quando nosso loop iterar (ou seja, i é igual a 3, e j é igual a 4), encontraremos o mesmo fantasma novamente. A posição à direita estará vazia, e o fantasma será movido para lá. O loop iterará novamente, j será igual a 5, e o mesmo fantasma está lá novamente. Perceba então que empurramos o mesmo fantasma várias vezes para a direita.

Precisamos encontrar uma maneira de saber se já empurramos aquele fantasma ou não. Podemos pensar em diversos algoritmos para isso, mas nossa solução será criar uma cópia do mapa. Com uma cópia, poderemos iterar em uma, e mover na outra. Assim, não encontraremos o mesmo fantasma duas vezes.

Vamos criar então uma função que copia um mapa. Essa função receberá então dois mapas: o destino e a origem. Copiar um mapa não é uma tarefa simples. Precisamos copiar os dois inteiros, linhas e colunas, alocar a matriz, e copiar linha a linha da matriz origem.

Para copiar uma linha, usaremos a função strcpy(). O nome já diz o que ela faz: copia uma string em outra. A função recebe dois parâmetros, "destino" e "origem". Essa função nada mais faz do que varrer o array e copiar o conteúdo na posição original na destino, e parar quando encontrar um NULL (\0).

Veja a função copiamapa(), dentro do mapa.c:

void copiamapa(MAPA\* destino, MAPA\* origem) {

destino->linhas = origem->linhas;

destino->colunas = origem->colunas;

alocamapa(destino);

for(int i = 0; i < origem->linhas; i++) {

strcpy(destino->matriz[i], origem->matriz[i]);

}

}

A função fantasmas() agora, ao começar, fará uma cópia do mapa m. Em seguida, varrerá cada caractere do mapa copia. Sempre que achar um fantasma, sabemos que ele também existe em m. Se for possível movê-lo (ou seja, ehvalida() e ehvazia() retorna verdadeiro para (i, j+1) no mapa m), o fazemos somente no mapa m.

Em seguida, como o fantasma não existirá na próxima posição à direita da copia (afinal, esse mapa está intocado), não teremos mais o problema do fantasma aparecer novamente na próxima iteração do loop.

Ao final, liberamos o mapa copia, afinal, fizemos uso de alocação dinâmica para criá-lo, e ele não é mais necessário.

Em código:

void fantasmas() {

MAPA copia;

copiamapa(&copia, &m);

for(int i = 0; i < copia.linhas; i++) {

for(int j = 0; j < copia.colunas; j++) {

if(copia.matriz[i][j] == FANTASMA) {

if(ehvalida(&m, i, j+1) && ehvazia(&m, i, j+1)) {

andanomapa(&m, i, j, i, j+1);

}

}

}

}

liberamapa(&copia);

}

É comum fazermos uso de estruturas auxiliares para nossos algoritmos. Nesse caso, em particular, criamos uma cópia do mapa para conseguirmos ter os valores originais mesmo após modificá-los.

Uma grande parte da vida do programador é criar **estruturas de dados** que facilitem a manipulação dos dados que temos. Esse é, aliás, um outro assunto para você estudar futuramente.

Usamos a função strcpy() para copiar um array para o outro. Essa função, como já explicamos, serve para copiar strings. Mas, às vezes, queremos copiar outros elementos que não strings. Por exemplo, podemos querer uma struct inteira.

Imagine o seguinte código:

struct contato {

char nome[20];

char email[50];

};

typedef struct contato CONTATO;

int main() {

CONTATO mauricio;

strcpy(mauricio.nome, "Mauricio Aniche");

strcpy(mauricio.email, "mauricio.aniche@caelum.com.br");

}

Se quiséssemos copiar a struct inteira, ao invés de copiar atributo por atributo, poderíamos fazer uso da instrução memcpy(). Como o nome já diz, ela copia o que está na memória. A assinatura dela é bem parecida com a da strcpy(), com a diferença de que, além de passar o destino e a origem, precisamos passar também a quantidade de bytes que queremos copiar.

No código abaixo copiamos o contato mauricio para o contato aniche. Repare que usamos sizeof() para calcular a quantidade de bytes da struct:

CONTATO aniche;

memcpy(&aniche, &mauricio, sizeof(CONTATO));

Outra função bastante comum quando manipulamos a memória é a função memset(). Geralmente a usamos para inicializar structs ou arrays com algum valor padrão (geralmente, "nulo"). Isso é especialmente útil, pois muitos compiladores de C não limpam as regiões de memória antes de liberá-la para o programador. Então é comum termos "lixo".

Para não termos nunca o problema de ter dados sujos, podemos sempre limpar nossas structs, por exemplo:

CONTATO marcelo;

// 0 significa NULO

memset(&marcelo, 0, sizeof(CONTATO));

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/905820c66a4f8e1a3a1c).

Nossos fantasmas ainda não são muito inteligentes: eles só sabem andar para a direita. O problema é que quando ele anda até a borda do mapa, ele fica parado lá para sempre. Vamos dar um pouco mais de inteligência pra eles. O fantasma agora, se não conseguir andar para a direita, tentará andar para alguma outra direção. A chance de ter uma direção válida é grande.

O que queremos fazer aqui é dar um pouco de **inteligência artificial**. Ou seja, ensinamos a máquina a tentar tomar a decisão que maximiza o ganho dela. Ou seja, o fantasma agora não ficará parado, ele andará sempre que possível em alguma direção. Claro que esse é o exemplo mais simples possível de inteligência artificial. Essa é uma área gigante de estudo e requer dedicação. Mas vamos devagar.

Faremos isso de maneira aleatória. Escolheremos, de maneira randômica, uma das 4 direções válidas. Se ela for válida, é para lá que o fantasma vai. Se ela não for válida, tentaremos novamente. Para que o programa não entre em um loop infinito, faremos isso no máximo 10 vezes. Se depois da décima tentativa, ainda não tivermos encontrado uma possível direção, então esse fantasma ficará parado nessa rodada.

Vamos começar então definindo uma estrutura auxiliar, que guardará as 4 possíveis direções. Veja que isso nos ajudará a não fazer 4 diferentes ifs: **"if sortedo é direita, tente ir pra lá"**, **"if sorteado é esquerda, tente ir para lá"**. E se andássemos em todas as direções possíveis, contando diagonais? Teríamos muito código. Pense sempre nisso. Muito código repetido (mesmo que um pouco diferentes um do outro) é problemático. Procure criar por estruturas que flexibilizem e facilitem seu código.

Começaremos por declarar então uma matriz dentro da nova função praondevaifantasma(), e já diretamente inicializá-la com as 4 direções possíveis. x, y+1 (direita), x+1, y (baixo), x, y-1 (esquerda) e x-1, y (cima):

int praondefantasmavai(int xatual, int yatual,

int\* xdestino, int\* ydestino) {

int opcoes[4][2] = {

{ xatual , yatual+1 },

{ xatual+1 , yatual },

{ xatual , yatual-1 },

{ xatual-1 , yatual }

};

}

Em seguida, criamos nosso loop de no máximo 10 iterações onde, dentro dele, sorteamos um número de 0 a 3 (a quantidade de linhas da matriz), e testamos essa direção. Se ela for válida, devolvemos a direção (repare que aqui precisamos receber ponteiros, pois queremos que a função devolva dois números), e fazemos a função retornar verdadeiro. Caso contrário, se as 10 tentativas falharem, retornamos falso. Não podemos esquecer de incluir time.h no arquivo:

int praondefantasmavai(int xatual, int yatual,

int\* xdestino, int\* ydestino) {

int opcoes[4][2] = {

{ xatual , yatual+1 },

{ xatual+1 , yatual },

{ xatual , yatual-1 },

{ xatual-1 , yatual }

};

srand(time(0));

for(int i = 0; i < 10; i++) {

int posicao = rand() % 4;

if(ehvalida(&m, opcoes[posicao][0], opcoes[posicao][1]) &&

ehvazia(&m, opcoes[posicao][0], opcoes[posicao][1])) {

\*xdestino = opcoes[posicao][0];

\*ydestino = opcoes[posicao][1];

return 1;

}

}

return 0;

}

Agora, basta mudarmos o algoritmo da função fantasmas(). Agora, ao encontrarmos um fantasma no mapa, invocaremos a função praondefantasmavai() para descobrir o caminho dele. Se a função nos retornar verdadeiro, sabemos que podemos movê-lo para aquela direção:

void fantasmas() {

MAPA copia;

copiamapa(&copia, &m);

for(int i = 0; i < copia.linhas; i++) {

for(int j = 0; j < copia.colunas; j++) {

if(copia.matriz[i][j] == FANTASMA) {

int xdestino;

int ydestino;

int encontrou = praondefantasmavai(i, j, &xdestino, &ydestino);

if(encontrou) {

andanomapa(&m, i, j, xdestino, ydestino);

}

}

}

}

liberamapa(&copia);

}

Pronto. Nossos fantasmas agora andam de maneira aleatória pelo mapa. Eles agora tem um pouco de inteligência na hora de andar. É claro que poderíamos melhorar ainda mais, por exemplo, fazendo os fantasmas sempre andarem na direção do herói, ou mesmo trabalhar em equipe, tentando cercá-lo. Esse é um bom desafio pra você.

Podemos deixar nosso código ainda mais legível, criando uma função para encapsular o teste do if. Veja que o condicional retorna verdadeiro se ehvalida() e ehvazia() retornam verdadeiro. Repare que ambas recebem inclusive os mesmos parâmetros. Podemos criar uma função, com um nome que explique melhor o que esse condicional significa, que invoca essas duas funções.

int podeandar(MAPA\* m, int x, int y) {

return

ehvalida(m, x, y) &&

ehvazia(m, x, y);

}

Dessa forma, o condicional ficará mais claro, e nosso código ainda mais fácil de ser reutilizado.

\*Está perdido? O código implementado até o momento pode ser encontrado [aqui]está em: <https://gist.github.com/mauricioaniche/390d17079d2d3df5abc0>.

## Acoplamento, encapsulamento e assinaturas de funções

O que falta agora é fazermos a finalização do jogo. O Foge Foge é um jogo onde não há vitória. Ele só acaba quando o herói morre. Em nosso jogo, o herói morrerá quando o fantasma "passar por cima dele".

Descobrir isso é fácil. Como sabemos que os fantasmas andam depois do personagem, se algum fantasma passar por cima do herói, então ao final do turno, não haverá herói no mapa. Já temos uma função que nos ajuda a encontrar algum personagem no mapa: a função encontramapa(). No entanto, precisaremos adaptá-la para nos avisar, caso o personagem não for encontrado.

Mudaremos então a assinatura da função, fazendo-a devolver agora um inteiro, que trataremos como booleano. Ela nos devolverá verdadeiro, caso encontremos o personagem, e falso, caso contrário:

int encontramapa(MAPA\* m, POSICAO\* p, char c) {

for(int i = 0; i < m->linhas; i++) {

for(int j = 0; j < m->colunas; j++) {

if(m->matriz[i][j] == c) {

p->x = i;

p->y = j;

return 1;

}

}

}

// não encontramos!

return 0;

}

Repare que nosso código continua compilando, mesmo após termos mudado a função. Isso porquê fizemos a mudança mais simples que podíamos na assinatura dela: trocamos de void para int.

Mas o quê aconteceria se trocássemos, por exemplo, o nome, ou mesmo os parâmetros que ela recebe? Nosso código pararia de compilar. Todo lugar que faz uso dela deixaria de funcionar, afinal a função mudou.

Não é fácil mudar a assinatura de uma função. Perceba o **acoplamento** entre a função e os diversos lugares que fazem uso dela. Pensar muito bem na assinatura dela é importantíssimo. Nossos programas até então são pequenos, mas em projetos maiores, mudar pode ser complicado.

Parte do trabalho do bom desenvolver é justamente pensar nos "contratos" que as funções tem. Por contrato, queremos dizer o nome da função (pois é por meio dele que você sabe o quê a função faz), os parâmetros que ela recebe e o que ela devolve. Se o contrato estiver bem definido, você raramente precisará mudar a assinatura.

A vantagem de ter pensado bem no contrato é que a implementação depois pode variar. Você reparou que em vários momentos do livro, mudamos o conteúdo da função? É o caso da função fantasma(). Alteramos o comportamento do andar dos fantasmas ao longo desse capítulo. Ou seja, mudamos nosso algoritmo para algo melhor, mas sempre "quebrar" o resto. Tudo continuou compilando.

Quando você estudar orientação a objetos, daremos a isso o nome de **encapsulamento**. Encapsular significa esconder os detalhes do algoritmo dentro da função, para que possamos mudá-los depois, sem nos preocuparmos em fazer o resto do sistema parar, disponibilizando para aqueles que querem fazer uso dela, um bom contrato que, por ser bom, a chance de mudança é baixa.

Como lá no começo do nosso jogo, pensamos bem que precisaríamos de uma função acabou(), agora basta preenchê-la com um algoritmo de verdade para decidir se o jogo acabou ou não. O que faremos é declarar uma estrutura POSICAO "inútil" e invocar a função encontramapa(), que procurará pelo herói. Se ela retornar 0, então a função acabou() retornar o oposto disso. Ou seja, se não encontrarmos o herói, acabou() retorna 1:

int acabou() {

POSICAO pos;

return !encontramapa(&m, &pos, HEROI);

}

Se jogarmos agora, veremos que estamos quase lá, mas ainda não está certo. Os fantasmas não conseguem passar por cima do herói. Além disso, o herói não consegue comer fantasmas. Precisamos melhorar nosso algoritmo.

O problema está na função podeandar(). Repare que o algoritmo lá era bem simplista. Ele permitia o personagem andar se a casa à direita estava vazia ou era vazia:

int podeandar(MAPA\* m, int x, int y) {

return

ehvalida(m, x, y) &&

ehvazia(m, x, y);

}

Essa função precisa ser mais inteligente. Podemos andar quando:

* A próxima posição está dentro dos limites do mapa (é exatamente o que a função ehvalida() faz).
* A próxima posição não é uma parede (precisamos criar essa função).
* A próxima posição não é um personagem idêntico ao que está andando (esse é o caso particular para que um fantasma não passe por cima de outro fantasma).

Vamos então criar a função ehparede() e deletar a função ehvazia(), pois não precisaremos mais dela. Acostume-se também com isso: se você não usa mais a função, apague-a. Código que não é usado só serve para complicar a vida do programador.

int ehparede(MAPA\* m, int x, int y) {

return

m->matriz[x][y] == PAREDE\_VERTICAL ||

m->matriz[x][y] == PAREDE\_HORIZONTAL;

}

Agora vamos criar uma função que compara o personagem que está na posição com o que foi passado por parâmetro. Apesar dessa função ter apenas uma linha, é muito melhor ter uma função dessa do que ter que ler o if e entender o que está acontecendo (lembre-se que o nome da função é importante, e nos ajuda a ler menos):

int ehpersonagem(MAPA\* m, char personagem, int x, int y) {

return

m->matriz[x][y] == personagem;

}

Por fim, vamos mudar a função podeandar(). Dessa vez, não teremos jeito: precisaremos mudar a assinatura dela para receber o personagem que está andando. Infelizmente precisaremos procurar por todos os lugares que invocam essa função e passar o novo parâmetro. Paciência, não pensamos nisso antes.

Repare que conseguimos ler o código dela do mesmo jeito que lemos a listagem anterior: **"pode andar se é válida E se não é parede E se não é personagem"**.

int podeandar(MAPA\* m, char personagem, int x, int y) {

return

ehvalida(m, x, y) &&

!ehparede(m, x, y) &&

!ehpersonagem(m, personagem, x, y);

}

Como mudamos a assinatura da função, vamos mudar também nos lugares que a invocam. Por sorte, temos apenas dois lugares: dentro da função move() e dentro da função praondefantasmavai(). Basta apenas passarmos a constante do respectivo personagem:

int praondefantasmavai(int xatual, int yatual,

int\* xdestino, int\* ydestino) {

// ...

if(podeandar(&m, FANTASMA,

opcoes[posicao][0], opcoes[posicao][1])) {

// ...

}

// ...

}

void move(char direcao) {

// ...

if(!podeandar(&m, HEROI, proximox, proximoy))

return;

// ...

}

Agora sim. Fantasmas podem passar por cima do herói, e o herói pode passar por cima dos fantasmas. Mas o que acontece se nosso herói passar por cima de todos os fantasmas? O jogo perde a graça. Vamos terminar o jogo também nesse caso. Isso é fácil, a função acabou() serve exatamente pra isso, e a ideia é análoga a que já temos: se não houver mais fantasmas no mapa, o jogo também acabou.

Repare aqui como criamos duas variáveis, perdeu e ganhou, apenas para aumentar a legibilidade do código. Essa é outra estratégia que você pode usar para facilitar a leitura: criar variáveis que explicam o retorno de uma função (ou uma conta, ou qualquer que seja o conteúdo que ela armazena):

int acabou() {

POSICAO pos;

int perdeu = !encontramapa(&m, &pos, HEROI);

int ganhou = !encontramapa(&m, &pos, FANTASMA);

return ganhou || perdeu;

}

Pronto. Os fantasmas agora andam de maneira aleatória, e podem matar o herói. O jogo está ficando quente.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/9164ca4d3a340586accb).

Nesta aula, aprendemos:

* Novamente, a extrair pequenas funções que são reutilizáveis.
* Que podemos (e devemos) criar estruturas auxiliares para facilitar a criação dos nossos algoritmos.
* Que podemos não abrir chaves no if, e que nesse caso, a execução do condicional será apenas a próxima instrução que aparecer.
* Que podemos copiar strings usando strcpy().
* Que podemos copiar ou limpar structs, arrays ou qualquer ponteiro, usando memcpy() e memset().
* Que precisamos pensar bem no contrato das nossas funções, pois mudá-los é difícil.
* Que podemos usar variáveis apenas para facilitar a legibilidade da nossa função.

A próxima funcionalidade do nosso jogo é dar a possibilidade do nosso herói comer a pílula do super poder. Se ele capturar essa pílula, apertando a tecla "B", ele soltará uma bomba que destruirá todos os fantasmas que estiverem a até 3 casas de distância, à direita.

Imagine o mapa abaixo, por exemplo, e suponha que W também seja fantasma. Todos os fantasmas anotados como W morrerão, caso o herói opte por soltar a bomba naquele momento. Perceba que todos eles estão há no máximo 3 casas de distância.

|--------|

|.....F..|

|.....-..|

|.@W.W...|

|.. ..FF.|

|.....F..|

|..F..F..|

|--------|

Vamos começar então pela lógica de capturar a pílula. Ela será representada pela letra P no mapa. Caso o jogador mova o herói em uma casa que a tenha, devemos guardar isso de alguma forma: a variável global tempilula nos dirá quantas pílulas o jogador tem para gastar. Vamos também inicializá-la com 0, afinal o jogador começa sem nenhuma pílula:

MAPA m;

POSICAO heroi;

// nova variável global

int tempilula = 0;

Agora, basta igualarmos essa variável a 1 sempre que o herói andar por cima de uma dessas pílulas. Para isso, na função move(), basta adicionarmos a verificação: a casa destino tem uma pílula? Se sim, então coloque verdadeiro nela:

void move(char direcao) {

// ...

if(!podeandar(&m, HEROI, proximox, proximoy))

return;

if(ehpersonagem(&m, PILULA, proximox, proximoy)) {

tempilula = 1;

}

// ...

}

Vamos até exibir para o jogador a quantidade de pílulas que ele tem naquele momento. Vamos colocar esse código dentro do método main() (embora não seja o local ideal, mas depois discutimos sobre isso):

int main() {

// ...

do {

printf("Pílula: %s\n", (tempilula ? "SIM" : "NÃO"));

imprimemapa(&m);

// ...

}

// ...

}

A próxima etapa é fazer nosso programa entender outras entradas que não uma direção. Ou seja, se a tecla pressionada for uma direção, o método move() deve ser invocado. Mas, se a tecla for um B, então ele deve fazer a bomba explodir.

Precisamos fazer o teste da tecla digitada agora no loop principal do programa e não mais dentro da função move(). Assim, nosso programa conseguirá dizer qual função deverá ser invocada. A constante BOMBA deve ser definida no header file:

do {

printf("Pílula: %s\n", (tempilula ? "SIM" : "NÃO"));

imprimemapa(&m);

char comando;

scanf(" %c", &comando);

if(ehdirecao(comando)) move(comando);

if(comando == BOMBA) explodepilula();

fantasmas();

} while (!acabou());

Vamos declarar a função explodepilula() apenas para nosso código compilar no momento, mas ainda sem uma implementação concreta, pois ela será o alvo da discussão da próxima seção:

void explodepilula() {

}

### Por que a main não é o melhor lugar?

Lembre-se de encapsulamento, e lembre-se que você deve fazer o programador ler a menor quantidade de código possível.

Idealmente a função main() apenas invoca e coordena o programa como um todo. Toda e qualquer regra de negócio deve estar encapsulada em uma função com um bom contrato. Ou seja, esse printf() mais o imprimemapa() deveriam estar encapsulados em uma função.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/ea6793804d9262097ed0).

Implementar a destruição dos fantasmas a direita é razoavelmente fácil. Basta fazermos um loop contado, 3 vezes, e verificar se a posição x, y+i (onde x e y vem da posição heroi) é válida e não é uma parede. Caso isso seja verdade, basta trocarmos aquela posição por vazio:

void explodepilula() {

for(int i = 1; i <= 3; i++) {

if(ehvalida(&m, heroi.x, heroi.y+i) &&

!ehparede(&m, heroi.x, heroi.y+i)) {

m.matriz[heroi.x][heroi.y+i] = VAZIO;

}

}

}

Podemos ainda tratar o caso da parede. Se tiver uma parede no caminho, então o efeito da bomba para de propagar para as outras casas. Ou seja, a parede é imune a bomba. Para isso, mudaremos um pouco a maneira com que nosso algoritmo foi desenhado: Assim que encontrarmos uma parede, quebramos o loop.

void explodepilula() {

for(int i = 1; i <= 3; i++) {

if(ehvalida(&m, heroi.x, heroi.y+i)) {

if(ehparede(&m, heroi.x, heroi.y+1)) break;

m.matriz[heroi.x][heroi.y+i] = VAZIO;

}

}

}

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/4f9f9cb84ce9d6196fcd).

Ótimo. Temos nosso algoritmo funcionando para uma direção. Mas, como sabemos, temos várias maneiras diferentes de implementar o mesmo algoritmo. Vamos tentar algo diferente agora. Sabemos que a principal parte dessa função é justamente a linha em que colocamos VAZIO na posição:

m.matriz[x][y] = VAZIO;

Vamos colocar somente isso então na nossa função, que receberá x e y como parâmetros:

void explodepilula(int x, int y) {

m.matriz[x][y] = VAZIO;

}

Sabemos também que depois de marcar a posição como VAZIO, o mesmo deve acontecer para a posiçõ y+1. É o que faremos: **chamaremos a função dentro dela mesma**, passando y+1 dessa vez. Sim, isso é possível, e chamamos isso de **função recursiva**. Ou seja, ela invoca ela mesma:

void explodepilula(int x, int y) {

m.matriz[x][y+1] = VAZIO;

explodepilula(x, y+1);

}

Como mudamos a assinatura dela, precisamos mudar a invocação dela também:

if(comando == BOMBA) explodepilula(heroi.x, heroi.y);

Pare e pense no que está acontecendo. Invocamos a função explodepilula() pela primeira vez dentro do método main. Lá, passamos heroi.x e heroi.y como parâmetros. Suponha que eles valham 1,1 nesse momento. A função então começa e limpa o que está na posição 1,2 (repare o +1 dentro da matriz). Em seguida, ela invoca ela mesma, com os parâmetros x, y+1, ou seja, 1, 2. Vamos para a segunda execução da função. Aqui, ela limpa o que está na posição, e se auto-invoca novamente, dessa vez, com os valores 1, 3. E assim por diante.

Você talvez tenha visto um problema nisso. Se não conseguiu perceber, rode a aplicação, e exploda a bomba. Você verá um erro:

Segmentation fault: 11

Você percebeu que a função nunca parou de invocar a si mesma? Ela se chama para 1,1, depois para 1,2, depois para 1,3, e repete isso até o infinito. Ou seja, temos uma chamada recursiva infinita. A função fica invocando ela mesma até que não haja mais espaço na máquina para guardar a pilha de execução. E aí, temos o tal do **segmentation fault**.

Quando criamos funções recursivas, precisamos deixar claro em que momento ela para de chamar ela mesma. E isso, claro, depende do problema que estamos resolvendo. Aqui, sabemos que precisamos executar a função 3 vezes. Para isso, colocaremos um "contador". Ou seja, uma variável qtd (de "quantidade"), que quando for 0, significa que a função deve parar de chamar a si própria. E como fazemos esse número ser zero? Na invocação recursiva, passamos qtd-1. Em código:

void explodepilula(int x, int y, int qtd) {

// se acabou o numero de vezes,

// entao acaba a função

if(qtd == 0) return;

m.matriz[x][y+1] = VAZIO;

// dessa vez, passamos qtd-1, pois

// já rodamos uma vez a função

explodepilula(x, y+1, qtd-1);

}

Vamos agora simular a execução dessa função. A função explodepilula() é invocada na main(). Vamos supor que os parâmetros passados para ela sejam 1, 1, 3. Na primeira execução, a função limpa a posição 1, 2, e se auto-invoca novamente com os parâmetros 1, 2, 2. A primeira execução é então suspensa para que a segunda execução aconteça. Nela, a posição 1, 3 é limpa, e ela se auto-invoca com os valores 1, 3, 1. A segunda invocação então é suspensa, para a terceira execução. A casa 1, 4 é limpa, e a a função é novamente invocada com os valores 1, 4, 0. Como qtd é igual a zero, a quarta execução morre. Com isso, a terceira execução volta a ativa e acaba, afinal não há mais nada nela. A segunda invocação então volta, e também acaba. O mesmo acontece com a primeira, finalizando a execução da função como um todo.

Acabamos de escrever nossa primeira função recursiva. Uma função recursiva é aquela que invoca a si mesma ao longo de sua execução. Para que isso funcione corretamente, ela precisa em algum momento parar de invocar-se (chamamos isso de **ponto de fuga**).

# Fibonacci

Podemos escrever muitas outras funções de maneira recursiva. A função matemática de Fibonacci é um bom exemplo. Ela é expressa por meio da seguinte fórmula:

F(0) = 0

F(1) = 1

F(n) = F(n-1) + F(n-2)

Pela natureza da função, perceba que ela invoca a ela mesma (ou seja, recursiva). Fibonacci de um número qualquer é igual à soma do Fibonacci do número anterior mais o Fibonnaci do número anterior ao anterior.

Então, escreva uma função recursiva que calcula o número de Fibonacci.

Podemos escrever então a seguinte função recursiva para calculá-lo. Repare que sempre precisamos de um ponto de fuga e, nesse caso, temos, afinal sabemos que Fibonacci de zero é 0, e Fibonacci de 1 é 1.

Em código, teríamos algo como:

int fib(int n) {

if(n == 0) return 0;

if(n == 1) return 1;

return fib(n-1) + fib(n-2);

}

Repare que a função invoca a si mesma passando sempre -1 e -2. Em algum momento, esses parâmetros serão iguais a 0 ou 1, e aí a função entra em seu ponto de fuga, e para de invocar a si mesma.

Um bom exercício é desenhar a árvore de invocações e entender bem o que acontece quando fazemos fib(5), por exemplo.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/ea529d890e3245ea4fcf).

Precisamos melhorar essa função. Afinal, ela ainda ignora possíveis exceções, como o caso de uma das 3 casas a direita não serem válidas ou serem paredes, por exemplo. Esses são somente outros pontos de fuga da função. Caso alguma dessas afirmações seja verdade, terminamos a execução dela. Em código:

void explodepilula(int x, int y, int qtd) {

if(qtd == 0) return;

if(!ehvalida(&m, x, y+1)) return;

if(ehparede(&m, x, y+1)) return;

m.matriz[x][y+1] = VAZIO;

explodepilula(x, y+1, qtd-1);

}

Agora, precisamos expandir a ideia para as quatro possíveis direções, e não só direita. Repare que a única coisa que muda de uma direção para a outra são as contas que fazemos com x e y. Para a direita, somamos 0,1. Para a esquerda, somaríamos 0,-1. Para cima, -1, 0, e para baixo, 1, 0.

Vamos então receber esses números como parâmetros da função. Assim, a função fica genérica o suficiente para servir a todos os casos. Dessa vez, faremos uso de variáveis auxiliares novox e novoy. Vamos inclusive renomear essa função para explodepilula2:

void explodepilula2(int x, int y, int somax, int somay, int qtd) {

if(qtd == 0) return;

int novox = x+somax;

int novoy = y+somay;

if(!ehvalida(&m, novox, novoy)) return;

if(ehparede(&m, novox, novoy)) return;

m.matriz[novox][novoy] = VAZIO;

explodepilula2(novox, novoy, somax, somay, qtd-1);

}

Renomeamos a função, justamente para criar a função explodepilula(), que fará as 4 invocações necessárias, para cada direção. É essa a função que será invocada dentro da main():

void explodepilula() {

explodepilula2(heroi.x, heroi.y, 0, 1, 3);

explodepilula2(heroi.x, heroi.y, 0, -1, 3);

explodepilula2(heroi.x, heroi.y, 1, 0, 3);

explodepilula2(heroi.x, heroi.y, -1, 0, 3);

}

Funções recursivas tem vantagens. Elas geralmente são mais simples e fáceis de serem lidas e implementadas. Mas, dependendo do caso, elas podem não ser tão performáticas. Afinal, sempre que invocamos uma função dentro da outra, o programa é obrigado a manter todas as variáveis que existem na primeira função em algum canto da memória, até que a segunda função invocada acabe, e a primeira volte a ser executada. Se tivermos, por exemplo, 1000 funções invocadas na sequência, a máquina é obrigada a guardar as variáveis das 999 chamadas anteriores, que ainda não acabaram. O mesmo não aconteceria se tivéssemos usado algum tipo de loop, por exemplo.

Podemos prolongar a discussão aqui. Muitos compiladores modernos conseguem reescrever a recursão no momento da compilação e transformá-la em um loop convencional, que ocupa menos espaço da pilha da execução. Mas esse não é o foco. O objetivo desse capítulo é fazer você entender o que são funções recursivas e como escrevê-las.

O último passo agora é deixar nosso jogo bonito. Afinal, ele não está nada apresentável. Como já sabemos que precisamos separar as responsabilidades ao máximo, vamos mover a função imprimemapa() para um novo arquivo, o ui.c (UI vem de **User Interface**, ou interface do usuário).

O primeiro passo é definirmos como imprimiremos cada personagem do mapa. Se abusarmos um pouco novamente de ASCII Art, podemos desenhar um fantasma de maneira muito mais elegante. Mas, precisaremos de mais de uma linha para isso. Vamos declarar então uma variável que guarda o desenho do nosso fantasma, em várias linhas. O desenho terá 4 linhas por 6 caracteres. Repare, no entanto, que declaramos 7 posições na segunda dimensão da matriz, justamente para guardar o \0 final:

char desenhofantasma[4][7] = {

{" .-. " },

{"| OO| " },

{"| | " },

{"'^^^' " }

};

A ideia é que quando encontrarmos um F no mapa, imprimiremos esse desenho. Quando encontramos um @, imprimiremos o herói, e assim por diante. Vamos criar então o desenho dos outros possíveis caracteres do nosso mapa:

char desenhoparede[4][7] = {

{"......" },

{"......" },

{"......" },

{"......" }

};

char desenhoheroi[4][7] = {

{" .--. " },

{"/ \_.-'" },

{"\\ '-." },

{" '--' " }

};

char desenhopilula[4][7] = {

{" "},

{" .-. "},

{" '-' "},

{" "}

};

char desenhovazio[4][7] = {

{" "},

{" "},

{" "},

{" "}

};

Com isso em mãos, precisamos agora imprimir os desenhos na hora certa. Mas, diferentemente do original, onde fazíamos dois loops aninhados (um para cada dimensão da matriz), aqui precisaremos ser um pouco mais inteligentes.

Repare que cada desenho tem 4 linhas. Como estamos usando a API simples de escrever na tela, não conseguimos escrever 4 linhas para desenhar o primeiro caractere que aparece no mapa, e depois voltar para a primeira linha novamente, para começar a desenhar o segundo caractere.

Precisamos ler a primeira linha do mapa, e imprimir as primeiras linhas de todos os personagens que estão ali. Depois todas as segundas linhas, depois todas as terceiras e, por fim, todas as quartas linhas. Só então poderemos começar a impressão da segunda linha do mapa.

Para isso, precisamos então de mais um loop contado, de 1 até 4 (ou 0 a 3), que imprimirá a linha certa de todos os desenhos. Veja o código abaixo. Temos a linha em i, a coluna em j e a parte do desenho a ser impressa em parte:

void imprimemapa(MAPA\* m) {

for(int i = 0; i < m->linhas; i++) {

for(int parte = 0; parte < 4; parte++) {

for(int j = 0; j < m->colunas; j++) {

// eh aqui que imprimiremos

}

}

}

Vamos criar uma função que apenas imprimirá a parte especificada do desenho. A assinatura dessa função deve ser imprimeparte(char desenho[4][7], int parte), pois precisamos receber a matriz de 4x7 e o inteiro que nos diz a parte:

void imprimeparte(char desenho[4][7], int parte) {

printf("%s", desenho[parte]);

}

Agora, basta fazermos a função imprimemapa() invocar a função imprimeparte() de acordo com o caractere que estiver em m->matriz[i][j] e o número da parte a ser impressa. Um simples switch resolve o problema.

Repare na sintaxe que usamos em PAREDE\_VERTICAL e PAREDE\_HORIZONTAL. Como em ambos os casos queremos imprimir o desenho desenhoparede, podemos colocar um case embaixo do outro, e dessa forma, ambos terão o mesmo comportamento:

void imprimemapa(MAPA\* m) {

for(int i = 0; i < m->linhas; i++) {

for(int parte = 0; parte < 4; parte++) {

for(int j = 0; j < m->colunas; j++) {

switch(m->matriz[i][j]) {

case FANTASMA:

imprimeparte(desenhofantasma, parte);

break;

case HEROI:

imprimeparte(desenhoheroi, parte);

break;

case PILULA:

imprimeparte(desenhopilula, parte);

break;

case PAREDE\_VERTICAL:

case PAREDE\_HORIZONTAL:

imprimeparte(desenhoparede, parte);

break;

case VAZIO:

imprimeparte(desenhovazio, parte);

break;

}

}

printf("\n");

}

}

}

No ui.c, não podemos esquecer de importar os header files necessários:

#include <stdio.h>

#include "mapa.h"

Com o algoritmo pronto, precisamos apenas acertar as arestas. Precisamos garantir que a função imprimemapa() não existe mais nem em mapa.c e mapa.h. Ela agora está em ui.h. Repare que ele precisa incluir mapa.h, afinal, ele faz uso de MAPA por lá:

#include "mapa.h"

void imprimeparte(char desenho[4][7], int parte);

void imprimemapa(MAPA\* m);

Precisamos incluir ui.h dentro de fogefoge.c, afinal é ele que invoca (já invocava, aliás) a função imprimemapa().

#include <stdio.h>

#include <stdlib.h>

#include "time.h"

#include "fogefoge.h"

#include "mapa.h"

// novo include

#include "ui.h"

O código está todo certo. Mas ao tentarmos compilar nossos 3 arquivos juntos (sim, agora são 3!), recebemos diversos erros parecidos com os abaixo:

> gcc fogefoge.c mapa.c ui.c -o fogefoge.out

In file included from fogefoge.c:6:

In file included from ./ui.h:4:

./mapa.h:12:8: error: redefinition of 'mapa'

struct mapa {

^

./mapa.h:12:8: note: previous definition is here

struct mapa {

^

O erro nos diz que estamos "redefinindo mapa". Ou seja, por algum motivo ele encontrou uma nova declaração da struct mapa em nosso código. Mas se temos certeza que isso não acontece, porquê o compilador se perdeu?

## Ifdefs e Ifndefs

O problema acontece porque quando fazemos um "include", o compilador literalmente inclui o código daquele arquivo ali. É como se fosse um copia-e-cola feito de forma automatica pelo compilador. E como agora temos uma mistura de inclusões, afinal o fogefoge.c inclui tanto mapa.h quanto ui.h, mas o ui.h também inclui mapa.h, acabamos por repetir a mesma declaração.

A primeira solução para isso é achar a combinação de inclusões a se fazer. Por exemplo, se sabemos que o ui.h inclui mapa.h, então só precisamos incluir o primeiro, pois o segundo já será incluído por ele. O problema é que essa solução pode ficar bastante complicada a medida que nosso programa cresce e torna-se mais complicado.

Precisamos de uma maneira de dizer ao compilador para não incluir um arquivo se ele já foi incluído. Ou para ignorar uma segunda declaração.

Similar as diretivas include e define, o compilador da linguagem C não dá outros 2 interessantes: o ifdef e o ifndef. Como o próprio nome diz, o que eles fazem são ifs. Mas são ifs diferentes: eles acontecem em tempo de compilação. O restante do nome, o "def", é porque o condicional é feito em cima de um "define".

Entender isso parece complicado. Mas em código é muito mais simples. Veja o código abaixo. Temos 3 printfs. No entanto, o segundo está entre um ifdef. Repare que a condição é se o #define IMPRIME existe. Como ele não existe, a saída desse programa é simplesmente: "antes do imprimiu" e "depois do imprimiu".

#include <stdio.h>

int main() {

printf("antes do imprimiu\n");

// esse trecho de código não será compilado,

// pois o compilador viu que a constante

// IMPRIME nunca foi definida

#ifdef IMPRIME

printf("imprimiu\n");

#endif

printf("depois do imprimiu");

}

No entanto, se definirmos uma constante IMPRIME, com qualquer valor (ou até mesmo vazio), no momento da compilação, o compilador perceberá que aquela constante existe, e compilará o que está dentro do ifdef:

// definimos a constante e,

// mesmo que vazia, ela existe!

#define IMPRIME

int main() {

printf("antes do imprimiu\n");

// o compilador percebe que ela existe,

// então compilará o que está dentro do ifdef.

#ifdef IMPRIME

printf("imprimiu\n");

#endif

printf("depois do imprimiu");

}

A grande charada aqui é perceber que esse teste é feito em tempo de compilação. E se o teste falhar, o compilador nem olhará para o que estiver dentro do ifdef: ele realmente ignorará, e não compilará o que estiver lá. Claramente, o ifndef é idêntico: a diferença é que ele verifica se a constante não foi definida.

Sabendo disso, a solução elegante para o problema das redefinições e includes por todo lado é usar essas diretivas de compilação dentro de nossos .h. A primeira coisa que faremos em cada um deles é perguntar se uma constante qualquer (por exemplo, \_UI\_H\_) existe. Se ela não existir, significará para nós que aquele header não foi incluído por nenhum outro arquivo. Então, dentro do ifndef, colocaremos todo o conteúdo que queremos, como declaração de funções e structs, e definiremos a constante \_UI\_H\_. A partir de agora, ela existe. Isso significa que a próxima vez que incluirmos esse header, a constante existirá, o ifndef será falso, e o compilador não passará novamente pelas mesmas definições de funções e struct, evitando o problema da redeclaração.

Em código, o ui.h ficaria:

#ifndef \_UI\_H\_

#define \_UI\_H\_

#include "mapa.h"

void imprimeparte(char desenho[4][7], int parte);

void imprimemapa(MAPA\* m);

#endif

O mesmo aconteceria com o mapa.h. Obviamente, daremos um nome diferente à constante que é usada:

#ifndef \_MAPA\_H\_

#define \_MAPA\_H\_

// todo o conteúdo aqui dentro

#endif

E fogefoge.h:

#ifndef \_FOGEFOGE\_H\_

#define \_FOGEFOGE\_H\_

// todo conteúdo aqui dentro

#endif

Perceba então que o compilador é poderoso. Podemos até dizer para ele o que deve ou não ser compilado. Na prática, usamos muito para evitar o problema da redeclaração de coisas que estão dentro de nossos header files. Assim, podemos incluir o mesmo arquivo várias vezes, e não nos preocupamos com detalhes.

Outro cenário onde diretivas de compilação são muito utilizadas é o caso de desenvolvimento de aplicativos para arquiteturas, plataformas ou sistemas operacionais diferentes. Todo o código que escrevemos nesse livro é ANSI C, ou seja, é padrão. Ele pode ser compilado em qualquer sistema operacional, como Windows, Linux e Mac, e o comportamento será o mesmo.

No entanto, temos muitas bibliotecas específicas de cada sistema operacional. Por exemplo, bibliotecas gráficas que só rodam no Windows, ou outras que só rodam no Linux. Nesses casos, muitos desenvolvedores optam por fazer grande parte do código em ANSI C, e nos pequenos pedaços específicos de cada sistema operacional, usar diretivas de compilação, para que trechos de código só sejam compilados no Windows, e outros trechos só no Linux. Também não entraremos em muitos detalhes dessa diferença. Mas acostume-se com eles: você pode encontrar um desses de vez em quando.

Pronto. Nosso jogo funciona, e temos agora uma interface muito mais amigável para nosso jogo de Foge Foge.

* O código final do jogo pode ser encontrado [aqui](https://gist.github.com/mauricioaniche/0aebab9a172ad7245a82).\*

Nesta aula, aprendemos:

* A entender o problema da redefinição de funções e structs que acontece quando nosso programa cresce em número de arquivos.
* A usar as diretivas de compilação #ifdef e #ifndef para resolver esses problemas.