Transcrição do Curso Alura C – Avançando na linguagem

Nosso próximo objetivo é desenvolver um jogo de forca. O computador escolherá, de maneira randômica, uma palavra que está salva em um arquivo de palavras, e o jogador deve chutar, letra por letra, até adivinhar a palavra. Se o jogador chutar 5 letras erradas, ele perde. Ao final, o computador oferecerá a possibilidade do usuário inserir uma nova palavra no banco de dados.

Nestes capítulos, a ideia é ensinar para você:

* Criar, varrer e manipular arrays de diferentes tipos.
* Usar arrays de chars (strings) e funções para manipulá-los.
* Entender e criar funções que recebem parâmetros e devolvem valores.
* Ler e escrever arquivos com formatos específicos.
* Aprender mais boas práticas de código, como a criação e extração de funções para evitar repetição de código.
* Criar header files.

**Como ele ficará?**

Nosso jogo também terá uma interface amigável. Ao iniciar, ele nos dará um boas vindas, e imprimirá a forca, por enquanto, sem ninguém:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/ Jogo de Forca \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_\_\_\_\_\_

|/ |

|

|

|

|

|

\_|\_\_\_

\_ \_ \_ \_ \_ \_ \_ \_ \_

Qual letra?

Se você chutar uma letra errada, ele o avisará e o homenzinho comecará a aparecer na forca:

Você errou: a palavra NÃO tem a letra X

\_\_\_\_\_\_\_

|/ |

| (\_)

|

|

|

|

\_|\_\_\_

\_ \_ \_ \_ \_ \_ \_ \_ \_

Qual letra?

Se você chutar uma letra certa, a letra aparece entre o vazado:

Qual letra? A

Você acertou: a palavra tem a letra A

\_\_\_\_\_\_\_

|/ |

|

|

|

|

|

\_|\_\_\_

\_ \_ \_ \_ \_ A \_ \_ A

Qual letra?

Se você perder, ele te enforcará por completo:

\_\_\_\_\_\_\_

|/ |

| (\_)

| \|/

| |

| / \

|

\_|\_\_\_

Uma caveira também aparecerá, avisando que o jogo acabou:

Puxa, você foi enforcado!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/ \

/ \

// \/\

\| XXXX XXXX | /

| XXXX XXXX |/

| XXX XXX |

| |

\\_\_ XXX \_\_/

|\ XXX /|

| | | |

| I I I I I I I |

| I I I I I I |

\\_ \_/

\\_ \_/

\\_\_\_\_\_\_\_/

Se você ganhar, um troféu aparecerá:

Parabéns, você ganhou!

\_\_\_\_\_\_\_\_\_\_\_

'.\_==\_==\_=\_.'

.-\: /-.

| (|:. |) |

'-|:. |-'

\::. /

'::. .'

) (

\_.' '.\_

'-------'

**É hora de começar!**

Está pronto para desenvolver esse jogo? **Mãos à obra!**

Para se começar um jogo de forca, de maneira análoga ao nosso jogo de adivinhação, é guardar a palavra secreta. Até agora, já conhecemos diversos tipos de variáveis, como int, double, float, long, mas ainda não vimos o tipo usado para se guardar um texto.

Em C, para guardarmos uma única letra, usamos o tipo char. E o declaramos da mesma forma que os outros. Mas, para se passar a letra a ser armazenada, usamos aspas simples. Para imprimir, usamos a máscara "c" no printf. Veja o código abaixo:

#include <stdio.h>

int main() {

char letra1 = 'M';

char letra2 = 'F';

char letra3 = 'A';

printf("%c %c %c", letra1, letra2, letra3);

}

Ele nos imprime as letras "M", "F" e "A". Mas e para guardar uma palavra? Será que precisamos armazenar letra a letra? A resposta é sim. Precisamos guardar uma a uma, mas para isso não teremos uma variável diferente para cada letra. Aliás, se tivéssemos, ficaríamos malucos. Imagina para guardar um texto inteiro? Quantas variáveis teríamos?

E o pior, não conseguiríamos fazer códigos dinâmicos para imprimir palavras de tamanhos diferentes. Não conseguimos escrever um for, por exemplo, para imprimir as várias letras:

// imagine uma palavra com 10 letras

for(int i = 1; i <= 10; i++) {

// não temos uma maneira de "gerar dinamicamente"

// o nome da variável a ser impressa.

// Ou seja, não conseguimos imprimir "letra1",

// depois "letra2", e assim por diante.

printf("%c", letra(i));

}

O que usaremos para isso é o que chamamos de **array**. Arrays são uma maneira de guardarmos "vários" de um só tipo. Esses elementos ficam armazenadas na memória, um ao lado do outro. Por exemplo, um array de inteiros guarda um monte de números inteiros. Um array de char guarda um monte de caracteres. E assim por diante. E a vantagem do array é que, no momento da declaração, dizemos o tamanho dele. E aí acessamos cada um desses elementos por meio de um índice.

Veja, por exemplo, uma declaração de um array de inteiros, com 5 posições. Veja que a declaração é idêntica, com exceção do uso dos colchetes:

int notas[5];

Agora, para usar esse array, guardando e lendo inteiros dele, basta usarmos os colchetes para definirmos a posição que queremos acessar. Afinal, temos 5. O único detalhe importante é que, se nosso array tem 5 posições, elas vão de 0 a 4. A primeira posição de um array é o 0 (e não o 1, como muitos confundem):

// guardando na primeira posição do array

notas[0] = 10;

// guardando na segunda posição

notas[1] = 5;

// imprimindo a primeira posicao

printf("%d", notas[0]);

Para então guardarmos nossa palavra secreta, vamos declarar um array de char. Vamos definir seu tamanho como 20:

char palavrasecreta[20];

Podemos guardar nossa palavra dentro dele, da mesma forma que estávamos fazendo antes. A diferença é que o número que passamos entre colchetes, pode vir de uma variável:

char palavrasecreta[20];

palavrasecreta[0] = 'M';

palavrasecreta[1] = 'E';

palavrasecreta[2] = 'L';

int numero = 3;

palavrasecreta[numero] = 'A';

palavrasecreta[4] = 'N';

palavrasecreta[5] = 'C';

palavrasecreta[6] = 'I';

palavrasecreta[7] = 'A';

Um trabalhão. Para imprimir a palavra, fica um pouco mais fácil. Afinal, como podemos usar variáveis ali no índice, conseguimos fazer um loop com o tamanho da palavra:

for(int i = 0; i < 8; i++) {

printf("%c", palavrasecreta[i]);

}

Entretanto, um array de char é um array especial. Como nós o usamos para guardar textos, temos funções especializadas para facilitar nossa vida na hora de colocar dados dentro dele. Essas funções também vão nos ajudar a escrever loops mais flexíveis (o 8 ali está fixo, mas queremos que ele se adapte ao tamanho da palavra).

A função sprintf() é muito parecida com a printf(). Enquanto o printf imprime na tela, a sprintf imprime em um array de chars (o **s\*\* é justamente de \*\*string**). Então, se quisermos guardar a palavra secreta no array, fazemos:

sprintf(palavrasecreta, "MELANCIA");

Para imprimirmos na tela, usamos o printf com a máscara "s". Ela faz com que todo o array seja impresso. Veja:

printf("%s", palavrasecreta);

Mas, veja que interessante. Nosso array tem 20 posições e a palavra "melancia" tem apenas 8 caracteres. Como que ele soube que deveria parar no último "a"? Por que ele não imprimiu 20 caracteres? Strings em C tem um pequeno detalhe: sempre que guardamos uma palavra lá, como melancia, que ocupa 8 posições, guardamos um caractere especial na 9a posição. Esse caractere nos diz que a string acaba ali, e que o resto é "lixo". Esse caractere é representado pelo símbolo \0 (barra zero). Apesar de serem dois caracteres, lembre-se que o barra é especial. Esse \0 é armazenado em uma única posíção.

Ou seja, quando fizemos o sprintf(), ele escreveu na verdade o texto "MELANCIA\0". A função printf(), sabendo disso, foi lendo o array de chars até encontrar o \0. Encontrou, parou.

Se estivéssemos declarando as letras dessa palavra uma a uma, igual fizemos acima, bastaria adicionar o \0 ao final, que o printf("%s") funcionaria normalmente:

char palavrasecreta[20];

palavrasecreta[0] = 'M';

palavrasecreta[1] = 'E';

palavrasecreta[2] = 'L';

palavrasecreta[3] = 'A';

palavrasecreta[4] = 'N';

palavrasecreta[5] = 'C';

palavrasecreta[6] = 'I';

palavrasecreta[7] = 'A';

// colocando o \0 para indicar que a string acabou

palavrasecreta[8] = '\0';

Vamos começar nosso jogo. Assim como o anterior, ele terá um loop principal. A ideia é que, enquanto ele não acertar a palavra, ou não morrer enforcado, continuamos a pedir uma letra pra ele. Vamos criar duas variáveis, uma chamada acertou, e outra chamada enforcou. Ambas serão booleanos, ou seja, guardarão 0 ou 1, indicando se o jogador ganhou ou foi enforcado, respectivamente.

Até então aprendemos dois tipos de loops: o for e o while. Ambos, antes de executarem o corpo de código que possuem, eles validam a condição. Isso quer dizer que se a condição do while for falsa desde o começo, o código não será executado nenhuma vez. Mas nesse nosso caso (e no jogo anterior também), ao menos uma vez precisamos rodar o bloco de código.

Podemos então fazer uso do do-while, um outro tipo de loop, idêntico ao while, com a exceção de que ele executa o bloco de código ao menos uma vez. Repare na condição while(!acertou && !enforcou). Conseguimos lê-la em português: **enquanto não acertou E não enforcou**.

int acertou = 0;

int enforcou = 0;

do {

// código aqui

} while(!acertou && !enforcou);

### Invertendo booleanos

Perceba então que o exclamação pode ser usado para "negar" (ou inverter) o valor de um booleano. Veja o código abaixo, onde declaramos dois inteiros, verdadeiro e falso, colocando 1 e 0 dentro deles. Em seguida, imprimos ambos duas vezes; uma vez normal, e a outra usando a exclamação para inverter o valor.

int verdadeiro = 1;

int falso = 0;

printf("%d %d", verdadeiro, !verdadeiro);

printf("%d %d", falso, !falso);

Perceba que o programa imprimirá 1 0 na primeira linha, e 0 1 na segunda. Ou seja, ele inverte o valor. Usamos o exclamação para conseguirmos escrever mais clara e sucintamente expressões booleanas.

O mesmo loop acima poderia ter sido feito usando a comparação com 0 (falso). Mas repare como escrevemos mais código dessa forma:

while(acertou == 0 && enforcou == 0);

Vamos agora capturar uma letra do usuário, e ver se essa letra existe na palavra secreta. Para isso, precisaremos fazer um for na palavra (afinal, sabemos o tamanho dela), e comparar se o char que está naquela posição bate com o informado pelo usuário. Para capturar o tamanho da palavra dentro de um array de chars, usamos a função strlen(). Ela está disponível em <string.h>.

A usaremos no for, que irá de 0 (primeira posição do array) até strlen(), que é a última posição preenchida. Em seguida, compararemos o conteúdo daquela posição com o chute dado, e avisaremos o usuário se ele acertou o chute:

do {

char chute;

printf("Qual letra? ");

scanf("%c", &chute);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(palavrasecreta[i] == chute) {

printf("A posição %d tem essa letra\n", i+1);

}

}

} while (!acertou && !enforcou);

A saída do programa, nesse momento, é a seguinte:

Qual letra? A

A posição 4 tem essa letra

A posição 8 tem essa letra

Qual letra? Qual letra? M

A posição 1 tem essa letra

Qual letra? Qual letra? E

A posição 2 tem essa letra

Qual letra? Qual letra? L

A posição 3 tem essa letra

Ainda longe do ideal, mas já é um começo.

Mas agora, repare com atenção, e veja que ele escreveu "Qual a letra?" duas vezes, várias vezes? Temos um bug aí. Isso aconteceu por um motivo bastante diferente. Veja que sempre que usamos o scanf(), digitávamos um número, por exemplo, e dávamos enter. O scanf é inteligente e sabe que "enter" não faz parte do número. Mas, agora que estamos lendo um caractere único, um "enter" é considerado um caractere. Então o scanf() se perde quando digitamos uma letra e um enter. O enter fica de "buffer", e quando executamos o scanf() pela segunda vez, ele nem nos pede a letra, ele passa o enter direto.

Para resolver isso, precisamos mudar um pouco a máscara. Ao invés de passarmos apenas o "%c", passaremos " %c". Repare no espaço. Ele diz ao scanf() para ignorar o enter. Isso resolve o problema:

scanf(" %c", &chute);

A saída agora, como esperado:

Qual letra? A

A posição 4 tem essa letra

A posição 8 tem essa letra

Qual letra? M

A posição 1 tem essa letra

Qual letra? E

A posição 2 tem essa letra

Qual letra? L

A posição 3 tem essa letra

*Está perdido? O código até este ponto pode ser encontrado* [*aqui*](https://gist.github.com/mauricioaniche/7d4243069caf65213031)*.*

**Laços encadeados**

Precisamos agora mostrar pro usuário a palavra, mas de maneira escondida. Ou seja, uma sequência de \_ \_ \_ \_, um underscore por letra. E, a medida que o jogador acerta uma letra, abrimos essa casa. Vamos começar devagar, imprimindo a palavra escondida.

O código é parecido com o loop anterior. Precisamos passar por cada letra da palavra secreta, e imprimir o underscore:

do {

for(int i = 0; i < strlen(palavrasecreta); i++) {

printf("\_ ");

}

printf("\n");

}

Agora, precisamos pensar em abrir a letra que ele já acertou. Para isso, precisamos guardar todos os chutes dele. E, para guardar muitos chars, você já sabe: array. Vamos declarar então um array responsável por guardar cada um dos chutes do usuário, e uma variável que conta o número de chutes já dados. Essa variável será especialmente importante, pois além de indicar a quantidade, ela nos indicará a posição desse array que o último chute dado deve ser guardado.

Por exemplo, imagine o array chutes e o inteiro tentativas, inicializado com 0. Quando tentativas for igual a 0, quer dizer que o usuário deu 0 chutes, e o primeiro chute deve ser guardado em chutes[0]. Quando tentativas for igual a 1, isso quer dizer que o usuário deu 1 chute, e o próximo chute deve ser guardado em chutes[1]. Quando quisermos ver todos os chutes, podemos também fazer um for que vai de 0 a tentativas.

Vamos começar declarando essas variáveis, antes do do-while:

char chutes[26];

int tentativas = 0;

Agora, vamos começar a salvar os chutes do usuário nesse array. Logo após capturarmos o char no scanf(), vamos guardá-lo no array. Já sabemos que a posição livre do array está na variável tentativas:

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

Agora vamos voltar ao nosso loop anterior, que exibe a palavra escondida. Precisamos de alguma inteligência ali: se o usuário já chutou aquela letra, precisamos exibir a letra. Se ele nunca chutou essa letra, exibimos o underscore. Para isso, para cada letra da palavra secreta, precisamos fazer um loop dentro do array de chutes, e ver se ela está lá.

Esse loop precisa ter uma variável de fora, tomando conta do resultado. Afinal, se acharmos a letra, em qualquer posição do array, devemos pará-lo, e imprimir a letra. Se, após passarmos em todo o array e o loop acabar, e a letra não está lá, então exibimos o underscore.

Veja, em código, a variável achou, que valerá 1, se acharmos a letra no array de chutes. Vamos colocar a impressão logo no começo do do-while, e remover o for que tínhamos mais embaixo (afinal, precisamos mostrar a forca antes de pedir a próxima letra). Repare também que usamos a letra j para o loop de dentro, afinal a variável i já está sendo usada pelo loop de fora:

do {

for(int i = 0; i < strlen(palavrasecreta); i++) {

int achou = 0;

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

if(chutes[j] == palavrasecreta[i]) {

achou = 1;

break;

}

}

if(achou) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

// ...

}

Veja que o for aninhado, ou seja, o for que está dentro do for, inicializa uma variável j que varia de 0 até a quantidade de tentativas já dadas. E aí, ele compara se a letra que está em chutes[j] é igual a letra que está em palavrasecreta[i]. Repare que j vai variar de 0 até o número de chutes, enquanto i vai valer 0. Depois, j vai variar novamente de 0 até o número de chutes, enquanto i vai valer 1. E assim por diante. Ou seja, compararemos todos com todos.

Vamos fazer o que chamamos de **teste de mesa**, ou seja, simular nosso código no papel. Imagine que a palavra secreta seja "MAO", e que os chutes dados foram "X" e "A" (ou seja, a quantidade de tentativas nesse momento é 2).

* i = 0, j = 0, palavrasecreta[0] é 'M' e chutes[0] = 'X'. M não é X, o loop continua.
* i = 0, j = 1, palavrasecreta[0] é 'M' e chutes[1] = 'A'. M não é A, o loop continua.
* i é incrementado.
* i = 1, j = 0, palavrasecreta[1] é 'A' e chutes[0] = 'X'. A não é X, o loop continua.
* i = 1, j = 1, palavrasecreta[1] é 'A' e chutes[1] = 'A'. **A é A**, o loop é quebrado, achou = 1, letra é impressa.
* i é incrementado.
* i = 2, j = 0, palavrasecreta[1] é 'O' e chutes[0] = 'X'. O não é X, o loop continua.
* i = 2, j = 1, palavrasecreta[1] é 'O' e chutes[1] = 'A'. O não é A, o loop continua.
* loop maior termina.

Loops dentro de loops podem parecer complicados no começo. Então use e abuse de testes de mesa. Coloque as variáveis que estão definidas naquele momento, e vá trocando uma a uma. Se quiser, coloque printfs no código, imprimindo as variáveis i, j, e quaisquer outras que quiser para ajudá-lo no entendimento.

Neste momento, nosso jogo já se parece mais com um jogo de forca. O usuário consegue chutar letras, e vamos mostrando as letras conforme ele acerta. Estamos caminhando bem.

Implemente o jogo de Forca. Chegue no mesmo ponto que chegamos neste capítulo.

O código ficará assim:

#include <stdio.h>

#include <string.h>

int main() {

char palavrasecreta[20];

sprintf(palavrasecreta, "MELANCIA");

int acertou = 0;

int enforcou = 0;

char chutes[26];

int tentativas = 0;

do {

for(int i = 0; i < strlen(palavrasecreta); i++) {

int achou = 0;

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

if(chutes[j] == palavrasecreta[i]) {

achou = 1;

break;

}

}

if(achou) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

} while (!acertou && !enforcou);

}

*Está perdido? Você pode ver* [*aqui*](https://gist.github.com/mauricioaniche/b0f02fbde64fef39ce95) *o código implementado até este momento.*

**Resumindo**

Nesta aula, você aprendeu:

* O que são arrays.
* Como declarar arrays.
* Como escrever dentro de arrays.
* Como pegar o conteúdo de uma posição do array.
* Que as posições do array começam no 0, e não no 1.
* Que arrays de char são arrays como qualquer outro.
* Que se guardamos uma string em um array de char, o caractere \0 deve ser colocado ao fim da palavra.
* Que podemos colocar loops dentro de loops.

Vimos até então que todas as variáveis que declaramos, sejam elas simples ou arrays, ficam armazenadas na memória do computador. A pergunta é: como ele faz isso? Como ele guarda por exemplo um número, dentro daquela peça de hardware que chamamos de memória?

Acredite ou não, uma memória de computador consegue apenas diferenciar duas coisas: se algo está imantado para um lado ou para o outro. A um dos lados atribuímos o valor 1, e ao outro, o valor 0. Ou seja, a memória de um computador não consegue guardar um número como "42", pois ela não consegue guardar nem "4" nem "2". Ela consegue guardar apenas "1"s e "0"s. E aí vem o desafio: precisamos representar qualquer coisa, usando apenas esses dois números.

Vamos ver como isso funciona devagar. Imagine o número 0, como representar ele? Fácil: 0. E o número 1? Também é fácil, 1. Pronto, temos um **bit**.

O desafio começa quando queremos representar o número 2, usando somente os números 0 e 1. A solução é usar o 10 para representar o número 2. Cuidado. Não leia 10 como dez, leia como "um zero", afinal ele não vale "dez". E o número três? 11. Isto é, "um um". E depois o número quatro? 100. Depois 101, 110, 111.

Ficamos com uma tabela:

0 => 0

1 => 1

2 => 10

3 => 11

4 => 100

5 => 101

6 => 110

7 => 111

8 => 1000

...

Repare que a conta aqui é parecida com a que fazemos com base decimal (que é a base que usamos, ou seja, representando todos os números possíveis usando apenas de 0 a 9). Veja que não temos um "desenho" para o número dez. O que fazemos quando chegamos no maior número da nossa escala (que no caso é 9)? Empurramos um dígito para esquerda, e começamos a contagem de novo: 10, 11, 12, 13, e assim por diante.

Com números binários, é a mesma coisa. Tínhamos o 0, e depois o 1. Como 1 é o maior dígito que temos, empurramos um dígito para a esquerda, e começamos a contagem de novo: 0, 1, 10, 11, 100, 101, 110, 111, e assim por diante.

Assim como na matemática tradicional, podemos preencher com zero a esquerda para padronizar a quantidade de dígitos. Por exemplo, se usarmos 8 dígitos, ou seja, 8 bits, temos:

0 => 00000000

1 => 00000001

2 => 00000010

3 => 00000011

4 => 00000100

5 => 00000101

6 => 00000110

7 => 00000111

8 => 00001000

9 => 00001001

10 => 00001010

11 => 00001011

12 => 00001100

13 => 00001101

14 => 00001110

15 => 00001111

O conjunto de 8 bits juntos é conhecido por 1 byte. Podemos seguir o mesmo padrão até qualquer número inteiro. Um inteiro em C tem 4 bytes. Isso quer dizer que usamos 4 vezes 8 = 32 bits. Com 32 dígitos, conseguimos representar números bastante grandes.

O conjunto de 8 bits juntos é conhecido por 1 byte. Podemos seguir o mesmo padrão até qualquer número inteiro. Um inteiro em C tem 4 bytes. Isso quer dizer que usamos 4 vezes 8 = 32 bits. Com 32 dígitos, conseguimos representar números bastante grandes.

## Bits: 8, 16, 32, 64

Se tivéssemos apenas oito bits para representar um número inteiro, poderíamos representar 2 elevado a 8 números, que é igual a 256. Parece pouco, mas acredite ou não, nossos primeiros videos games, como o Master System, tinham processadores de 8 bits, e sabiam só representar números de 0 a 255.

O Mega Drive, um pouco melhor, tinha um processador de 16 bits, e com isso, representava números inteiro de 0 a 65.535 (ou seja, 2 elevado a 16 números diferentes). Não é a toa que os jogos do Master tinham em geral no máximo 256 cores, enquanto as do Mega tinham até 65 mil.

Mas mesmo esse número, será que ele é capaz de representar todos os números inteiros que queremos? Ou ainda um processador capaz de entender e armazenar 16 bits é capaz de representar todas as letras de todos os alfabetos do mundo?

Os processadores modernos são capazes de representar números com até 64 bits, o que dá bons bilhões, além de todo o conjunto de caracteres dos alfabetos existentes no mundo. Mesmo assim existem otimizações para representar o alfabeto, como um padrão famoso por ter substituído o ASCII como mais utilizado na internet em 2008, o UTF-8.

Agora temos a questão dos números com ponto flutuante. Como representar tais números, se o computador só conhece zeros e uns? Do mesmo jeito que fizemos com as letras, precisamos de algum padrão. Para entender melhor, vamos criar o nosso próprio padrão, mais simples do que um usado hoje em dia.

Imagine o número treze e meio, também conhecido como 13,5. Quantos dígitos ele possui antes da casa decimal? Dois (em binário, 0010). Quaia são esses dígitos? 1 (0001) e 3 (0011). Por fim, o valor da casa decimal é 5 (0101). Portanto, para representar o número 13,5, utilizaremos a notação que indica o número de casas antes da vírgula (2), o valor dela, dígito a dígito (1 e 3) e o valor decimal (5):

0010 0001 0011 0101

Pronto. O computador é capaz de entender números como 13,5. Com isso, somos capazes de representar qualquer número com casas decimais, cuja quantidade de dígitos é finita, mesmo que essa técnica esteja longe de ser ótima ou perfeita.

Mas ainda temos uma limitação: neste tipo de abordagem, como representar o valor 1 / 3, isto é, um terço? Esse número tem uma quantidade infinita de dígitos.

Uma primeira opção seria arredondarmos o número 0,3, perdendo precisão:

0001 0000 0011

Já que o arredondamento foi ruim, podemos tentar melhorá-lo 0,33:

0001 0000 0011 0011

Ou mais ainda, 0,333:

0001 0000 0011 0011 0011

Como o computador costuma seguir padrões de tamanho fixo, é comum que o número de ponto flutuante, assim como um número inteiro, utilize 64 bits. Isso limita a nossa aproximação. Para nosso um terço, usando essa idéia apresentada (não ideal), temos a melhor aproximação de 0,333.3 com 62 casas após a vírgula.

0001 0000 0011 0011 0011 ... 0011

Nossa abordagem de representação é bastante limitada, mas a representação escolhida por um processador é bem mais inteligente que essa e permite aproximações muito boas.

A base binária, como vimos, tem somente dois algarismos distintos para representar **todos** os números. A que utilizamos no dia a dia, tem dez algarismos. Entretanto, uma outra base bastante utilizada, a base hexadecimal, que tem 16 algarismos. Para representar números de 0 a 15, precisamos apenas de quatro bits.

Agora precisamos de 16 "desenhos" para representar os 16 possíveis algarismos. Para isso completamos os números de 0 a 9 com as letras A, B, C, D, E e F para representar os outros seis algarismos faltantes. Novamente, sem a necessidade de decorar a fórmula de transformação, a tabela a seguir mostra os 16 primeiros números em base hexadecimal:

0 => 0

1 => 1

2 => 2

3 => 3

4 => 4

5 => 5

6 => 6

7 => 7

8 => 8

9 => 9

10 => A

11 => B

12 => C

13 => D

14 => E

15 => F

E poderíamos continuar, com:

16 => 10

17 => 11

18 => 12

...

32 => 21

...

177 => B1

...

255 => FF

Portanto, com dois algarismos de base hexadecimal (16 possibilidades cada), conseguimos representar 16 \* 16, 256, números.

Como representar letras (chars) em números binários? Basta fazermos uma tabela, dando um número para cada letra. Por exemplo, 'A = 65'. Então 'B = 66', 'C = 67':

A => 65 => 01000001

B => 66 => 01000010

C => 67 => 01000011

...

Z => 90 => 01011010

Pronto, o computador é capaz de representar todas as letras existentes no nosso alfabeto usando somente 8 dígitos, 0 ou 1. Essa tabela do nosso alfabeto, incluindo diversos outros caracteres é a **tabela ASCII**, usada por muito tempo como o principal padrão de tradução de números e caracteres por um computador.

Agora, um último desafio: como representar uma imagem? É fácil. Mapearemos uma imagem para um conjunto de números inteiros. E já sabemos que conseguimos representar números inteiros de forma binária.

Imagine então uma foto. Ela é composta por diversos pontos, e cada um desses pontos tem uma cor diferente. Imagine que a foto tem 1000 por 1000 pontos. O que podemos fazer é primeiro enumerar todas as cores existentes. Mas vamos limitá-las a 256 diferentes:

cor0 = verde

cor1 = vermelho

cor2 = azul

cor3 = preto

cor4 = branco

...

cor255 = cinza

Mas não podemos usar o nome da cor: precisamos dar um número para ela. Por exemplo:

cor0 = 00FF00 (verde)

cor1 = FF0000 (vermelho)

cor2 = 0000FF (azul)

cor3 = 000000 (preto)

cor4 = FFFFFF (branco)

...

cor255 = CCCCCC (cinza)

Com essa tabela em mãos, podemos escrever uma linha para cada ponto na foto. Se o primeiro ponto da foto é vermelho (cor1) e o segundo azul (cor2) e o terceiro cinza (cor255), em uma imagem com diversas linhas, temos:

cor1 cor2 cor255 cor3 cor3 cor2 cor3 cor255

cor1 cor3 cor255 cor3 cor2 cor3 cor3 cor255

...

Mas como sabemos que temos no máximo 256 cores, podemos representá-las mais facilmente com números em hexadecimal:

01 02 FF 03 03 02 03 FF

01 03 FF 03 02 03 03 FF

...

Isto é, a sequência 01 02 FF representa uma pequena imagem de 3 pontos, cujas cores seriam vermelho, azul e cinza. Já 0102FF03030203FF\n0103FF03020303FF representa uma imagem com 2 linhas e 8 colunas.

Com isso podemos representar qualquer imagem com tamanho finito (que caiba na memória) e que o número de cores esteja limitado a 256. Esse é o raciocínio atrás de um mapa de bits de cores, um **bitmap**, o formato BMP, muito utilizado antigamente no mundo da computação. Sua adoção não é mais tão grande devido ao tamanho que o mesmo ocupa: uma imagem com grande qualidade exige uma tabela de cores enorme, e um número de pontos maior ainda. Algoritmos de compressão como o JPEG dominam esse mercado.

Assim como pensamos na alternativa de representar um terço como uma fração (um dividido por três), poderíamos representar uma imagem como uma sequência de traços, círculos etc. E essa também é uma idéia boa para diversos tipos de imagem, e é o raciocínio por trás das imagens vetoriais, como o **SVG**.

O mesmo pode ser aplicado ao mundo da música, com o WAV e MP3, aos filmes etc. No fim, representamos desde um número inteiro até uma imagem de tomografia com apenas zeros ou uns. Assustador.

Você reparou que nosso jogo ainda não tem uma mensagem de bem vindo ao usuário? Vamos colocá-la então:

int main() {

// imprime cabecalho

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

// codigo continua aqui

}

Nosso jogo, em termos de funcionalidades, está andando bem. Mas vamos dar uma pausa nelas, para atacar outro detalhe importantíssimo na hora de fazermos um software. Você lembra do código do jogo de adivinhação? Ele ficou muito legal, mas ao final da implementação, tínhamos uma função main() com mais de 100 linhas de código. São muitas linhas, e isso dificulta muito o entendimento do código por outras pessoas. Você mesmo, faça o teste. Daqui algumas semanas, volte e olhe o mesmo código: você lembrará muito menos dele.

Precisamos aprender a dividir nosso programa em partes menores, que sejam mais fáceis de serem lidas. Precisamos fazer igual as bibliotecas do C fazem. Por exemplo, quando você quer imprimir algo na tela, você chama a função printf(). Você não sabe como ela funciona por dentro, apenas sabe que ela imprime. A mesma coisa com o `scanf()´: você não sabe como ela faz, apenas o quê ela faz.

## Escrevendo funções

Imagine se conseguíssemos trocar então as 3 linhas responsáveis pela abertura do jogo, e trocar por uma única linha? Ao parecido com o comentário, mas em formato de código, sem espaços, usando os parênteses, e terminando a linha com ponto-e-vírgula:

abertura();

O que queremos aqui é criar o que chamamos de **funções**. Funções são blocos de código encapsulados e que podem ser reutilizados ao longo do nosso programa. Por exemplo, se criarmos uma função que se chama abertura(), contendo aquelas 3 linhas de impressão, sempre que a invocarmos, nosso programa fará isso. É igual ao printf() ou scanf(): ambos são funções, que utilizamos ao longo do nosso programa.

Vamos criar essa função e usá-la em nosso programa. Escreveremos esse código fora da função main (afinal, ela própria é uma função, e não podemos ter funções dentro de funções). Para declarar uma função, precisamos no mínimo de um nome. Pode ser qualquer palavra, como por exemplo, o nome que escolhemos. Então, faremos void abertura(). Você não precisa entender o void por enquanto, e nem o abre e fecha parênteses. Em seguida, abriremos chaves e colocaremos o conteúdo dela dentro desse bloco de código:

void abertura() {

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

}

Agora vamos fazer uso dessa função. E o faremos dentro do método main(). Vamos remover aquelas 3 linhas, que agora estão já na função certa, e apenas invocá-la. Para invocar é fácil, usamos seu nome e parênteses:

int main() {

abertura();

// codigo continua aqui

}

Se rodarmos nosso código agora, o comportamento ainda é o mesmo. A diferença é que agora nosso método main é um pouco mais fácil de ser lido. Quando o programador lê abertura(), ele sabe que é ali que é feita a abertura, mas não sabe os detalhes. Se ele quiser saber os detalhes, ele vai ler então o corpo da função abertura(). Diminuímos a quantidade de código que o programador precisou ler para entender o método main().

Vamos entender o que acontece com a máquina ao executar esse código. Quando o programa começa, ele automaticamente executa a função main(). Ele vai executando linha a linha do código. Até aí, sem novidades. Mas, quando ele encontra uma chamada para outra função, por exemplo, a função abertura(), ele então "suspende" a execução do método main(), e vai para o outro método. Lá, o procedimento é o mesmo. Ele executa linha a linha. Quando o método abertura() acaba, ele volta então para o método main e continua a execução a partir dali.

Se tivéssemos uma outra chamada de função dentro do abertura(), o comportamento seria o mesmo. A máquina suspenderia a execução desse método, executaria o outro, e quando ele acabasse, voltaria para ele, que, por sua vez, quando acabasse, voltaria para o main.

## Mais funções

Veja agora o código principal do nosso jogo, e mais especificamente o conteúdo dentro do do-while. Repare nos três comentários de código. Será que também não conseguimos os transformar em funções?

do {

// imprime a palavra secreta

for(int i = 0; i < strlen(palavrasecreta); i++) {

int achou = 0;

// a letra já foi chutada?

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

if(chutes[j] == palavrasecreta[i]) {

achou = 1;

break;

}

}

if(achou) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

// captura um novo chute

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

} while (!acertou && !enforcou);

Repare o primeiro comentário, **"imprime a palavra secreta"**. Ele engloba o for, que é responsável por imprimir a forca. O segundo comentário, **a letra já foi chutada?**, engloba o segundo for, que é responsável por dizer se uma determinada letra já foi chutada pelo usuário ou não. Por fim, o último comentário, **captura um novo chute**, pega um novo chute do usuário.

Vamos pegar então o último comentário e também transformá-lo em uma função. Afinal, isso deixará nosso código ainda mais legível. O procedimento é o mesmo: vamos criar a função chuta():

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

}

Vamos agora também invocá-la dentro do main. E, claro, no lugar do código antigo, que removemos em prol dessa nova função:

int main() {

do {

// ...

chuta();

} while(!acertou && !enforcou);

}

Fizemos o mesmo procedimento de anteriormente, mas dessa vez o código não compila. Veja o erro: ele reclama que a variável chutes não existe. E o compilador está certo, afinal a variável chutes existe apenas dentro da função main:

forca.c:15:2: error: use of undeclared identifier 'chutes';

did you mean 'chute'?

chutes[tentativas] = chute;

^~~~~~

chute

Variáveis vivem, e são visíveis, apenas no escopo em que foram declaradas. Por exemplo, se declaramos uma variável dentro do método main(), essa variável estará visível somente dentro desse método e dos escopos declarados dentro desse método (por exemplo, se você tiver um for dentro desse método, ele conseguirá ver essa variável).

No código abaixo, a variável multiplicador, declarada dentro do método main() é visível em todo lugar dentro dessa função. Já a variável resultado, declarada dentro do for é visível somente dentro desse for (ou em outros escopos que fossem declarados dentro dele). Entretanto, ambas não são visíveis dentro da função abertura():

void abertura() {

// aqui nao enxergamos a variável multiplicador

printf("Tabuada do ");

printf("Quero imprimir o multiplicador aqui, mas nao consigo...");

}

int main() {

int multiplicador = 2;

abertura();

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

int resultado = multiplicador \* i;

printf("%d x %d = %d", i, multiplicador,

resultado);

}

}

Para resolver esse problema, temos duas soluções. A primeira delas é passando as variáveis que queremos para as outras funções. E podemos fazer isso. Basta, no momento da declaração da variável, dizermos quais serão os parâmetros que ela receberá. E, na hora de invocarmos, passamos os parâmetros pra ela. Fazemos isso nos "parênteses" no momento da declaração. Basta dizermos o nome do parâmetro e seu tipo.

Veja, por exemplo, a função abertura() recebendo como parâmetro um inteiro multiplicador. Esse parâmetro é como se fosse uma variável para aquela função. Ela existe naquele escopo e pode ser manipulada como qualquer outra variável:

void abertura(int multiplicador) {

// aqui temos uma variável chamada multiplicador

// e seu valor será definido por quem chamar essa função

printf("Tabuada do %d", multiplicador);

}

Isso significa que não podemos mais invocar a função abertura() sem passar os seus parâmetros. Sempre precisamos passar os parâmetros que as funções requerem. Nesse caso em particular, precisamos sempre passar um inteiro pra ela. Veja:

// passando um inteiro diretamente

abertura(2);

E, já que ela recebe um inteiro, porque não passar também um inteiro que está em uma variável? Veja:

int main() {

// passando uma variável

int numero = 10;

abertura(numero);

}

Repare que estamos passando a variável numero como parâmetro para a função abertura(). Ela contém o número 10. Isso quer dizer que, quando a função abertura for executada, o parâmetro multiplicador valerá 10 também. É como se a máquina "copiasse" o valor da variável numero e colasse na variável multiplicador.

Para que esse nosso pequeno código de tabuada funcione, basta então invocarmos o método abertura(), passando a variável multiplicador. Repare que, apesar de ambos os nomes serem iguais (o da variável e o do parâmetro), não é o nome que fez tudo acontecer. Ou seja, o parâmetro multiplicador é diferente da variável multiplicador que está na main(); apenas os conteúdos é que são iguais. Lembre-se que, quando você chama uma função, por exemplo, abertura(10), a máquina pega o número 10 e coloca dentro do parâmetro que foi declarado na função (que pode ter qualquer nome).

void abertura(int multiplicador) {

printf("Tabuada do %d\n", multiplicador);

}

int main() {

int multiplicador = 2;

abertura(multiplicador);

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

int resultado = multiplicador \* i;

printf("%d x %d = %d\n", i, multiplicador,

resultado);

}

}

Está perdido? O código do exemplo da tabuada pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/84aa107071d8b8bf5fcf).

Escreva uma função potencia() que receba dois inteiros, a e b, calcule a potência a^b, ou seja, a elevado a b e imprima o resultado.

Inicializamos uma variável resultado com 1, e aí iteramos b vezes em um for, cujo código é multiplicar o número atual por a.

void potencia(int a, int b) {

int resultado = 1;

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

resultado = resultado \* a;

}

printf("O resultado é %d", resultado);

}

Voltando ao nosso jogo, precisamos fazer com que a função chuta receba então o array de chutes e a variável tentativas. Vamos então colocar esse parâmetro na assinatura do método. Repare em como declaramos que o parâmetro chutes é do tipo array de char.

void chuta(char chutes[], int tentativas) {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

}

E agora, passamos o array e o inteiro, que foram declarados no método main, para a função chuta():

chuta(chutes, tentativas);

tentativas++;

Novamente, repare que os parâmetros chutes e tentativas são "variáveis diferentes" do chutes e tentativas que estão no método main. Mas isso pode ser um problema, dependendo do caso. No próximo capítulo, falamos mais sobre isso.

Escreva uma função soma(int numeros[10]) que receba um array de inteiros e imprime a soma de todos os elementos dentro desse array.

Confira o resultado a seguir.

Segue uma possível implementação:

void soma(int numeros[10]) {

int total = 0;

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

total += numeros[i];

}

printf("O total é %d", total);

}

O código ficará assim:

#include <stdio.h>

#include <string.h>

char palavrasecreta[20];

char chutes[26];

int tentativas = 0;

void abertura() {

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

}

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

}

int jachutou(char letra) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

void desenhaforca() {

printf("Você já deu %d chutes\n", tentativas);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i])) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

void escolhepalavra() {

sprintf(palavrasecreta, "MELANCIA");

}

int main() {

int acertou = 0;

int enforcou = 0;

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

tentativas++;

} while (!acertou && !enforcou);

}

Para provar que quando usamos parâmetros em funções e passamos uma variável como parâmetro, mesmo que os nomes sejam iguais, elas são diferentes, veja o progama abaixo. O método main contém uma variável tentativas, e ele faz operações de incremento com essa variável e imprime. Em seguida, ele passa a variável para a função joga(), que faz um incremento, e imprime. Ao final, o método main imprime novamente o valor da variável.

#include <stdio.h>

void joga(int tentativas) {

printf("joga %d\n", tentativas);

tentativas++;

printf("joga %d\n", tentativas);

}

int main() {

int tentativas = 0;

printf("main %d\n", tentativas);

tentativas++;

printf("main %d\n", tentativas);

tentativas++;

printf("main %d\n", tentativas);

joga(tentativas);

printf("main %d\n", tentativas);

}

O interessante é a saída desse programa. Veja que a variável no método main() vale 2, quando chamamos a função joga(). Em seguida, a função joga() incrementa 1, e podemos ver que ela imprime o número 3. Mas, quando voltamos para a função main, ele imprime novamente 2:

main 0

main 1

main 2

joga 2

joga 3

main 2

Isso nos prova que ambas as variáveis apesar de terem o mesmo nome, são diferentes. Quando invocamos a função joga(), passando a variável tentativas, declarada no método main, a máquina copia o conteúdo dela, cria uma nova variável chamada tentativas, que será visível só na função joga(). Quando a função joga() termina, e o programa volta pro método main, a variável tentativas continua valendo 2. Afinal, ela não foi modificada depois disso.

Esse isolamento entre as funções é uma coisa boa. Imagine se você passasse uma variável como parâmetro para uma função, e essa função tivesse a liberdade de mudá-la? Seu programa seria mais complicado de ser entendido, afinal, você não saberia bem quem mudou a variável e quando. É por isso que chamamos isso de **passagem por cópia**. Ou seja, sempre que passamos uma variável para uma função, a máquina copia a variável e cria uma nova, para que a original fique intacta.

Muitas vezes a passagem por cópia é suficiente. Mas, às vezes não. Volte ao nosso jogo, e veja a variável tentativas. Ela deveria contar a quantidade de chutes já dados pelo jogador. O método chuta é o responsável por incrementar essa variável, mas é a variável tentativas do método main também fosse incrementada. Nesse momento, o parâmetro tentativas é o que é incrementado, e sabemos que eles são variáveis diferentes. Ou seja, gostaríamos, nesse caso, que a variável tentativas da função chuta fosse a mesma da função main.

void chuta(char chutes[26], int tentativas) {

char chute;

scanf(" %c", &chute);

chutes[tentativas] = chute;

}

int main() {

// ...

chuta(chutes, tentativas);

// a soma precisa acontecer na variável

// 'tentativas', que existe no escopo

// da função 'main'

tentativas++;

// ...

}

**Passagem por referência**

Podemos dizer ao nosso programa que não queremos que ele passe o parâmetro por cópia, mas sim, pelo que chamamos de **referência**. A passagem por referência é algo bastante avançado. Como que a máquina faz para passar uma variável por referência? Simples, ela passa a posição da memória que essa variável está. Imagine a memória do computador como uma fita imensa e numerada. Todas as variáveis de nossos programas ficam nessa fita, em algum lugar.

Pense novamente na passagem por cópia. Imagine que a variável tentativas do método main está na posição 10 da memória. Quando passamos por cópia, a máquina aloca um novo espaço na memória para a nova variável tentativas. Imagine que ela ficou na posição 200. Ou seja, uma variável aponta para a posição 10, enquanto a outra aponta para a posição 200.

Você precisa pensar agora em variáveis como se elas fossem setas, ou melhor, **ponteiros** para endereços de memória, pois é exatamente isso que elas são.



E, se no fundo, variáveis são apenas ponteiros para endereços de memória, por quê não ter duas variáveis que apontam para o mesmo endereço? Podemos fazer isso. Por exemplo, se quisermos ver o endereço de memória que uma variável aponta, usamos o caractere **&**. O programa abaixo imprime o endereço de memória da variável c. Repare que sempre que o rodarmos, o número será diferente, afinal, cada hora a máquina decide pegar um endereço de memória diferente (e que está livre naquele momento, claro):

#include <stdio.h>

int main() {

int c = 10;

printf("%d\n", &c);

}

Isso quer dizer que se passarmos como parâmetro para uma função, um &c, por exemplo, estaremos passando o endereço de memória que essa variável está. Isso quer dizer que se modificarmos o conteúdo que está dentro desse endereço, então ele será modificado para todo mundo que tenha uma variável apontada pra lá.

Se você declara uma variável do jeito que está acostumado, então a máquina esconde de você a ideia de que ela é na verdade um ponteiro. E é por isso que você consegue fazer coisas como atribuições, por exemplo, c = 10, e a máquina faz o processo de pegar o endereço de memória que está escondido em c e colocar o valor 10.

Mas, você também tem a opção de manipular diretamente um endereço de variável. Ou seja, declarar um ponteiro. Um ponteiro simplesmente contém um endereço de memória. Declarar um ponteiro é parecido com declarar uma variável: precisamos de um nome e de um tipo. Mas, completamos o tipo com estrela (\*).

Veja o exemplo abaixo, onde declaramos um ponteiro para um inteiro. Se ele é um ponteiro, quer dizer que podemos guardar o endereço da variável c lá dentro. Para pegar o endereço dela, basta usarmos o &.

int c = 10;

int\* ponteiro;

ponteiro = &c;

Nesse momento, a variável ponteiro aponta para o mesmo lugar que c aponta. Se imprimirmos então &c e ponteiro, o resultado é o mesmo:

printf("%d %d", ponteiro, &c);

Agora, se quisermos acessar o que está dentro do endereço para qual ponteiro aponta, usamos o \* na frente da variável. Por exemplo, se imprimirmos a variável c e \*ponteiro, o resultado é o mesmo (10):

printf("%d %d", \*ponteiro, c);

Ou seja, se a variável é um ponteiro, e queremos acessar o conteúdo que está dentro do endereço que ela aponta, usamos estrela. Veja novamente o código inteiro, e perceba como manipulamos uma variável normal e uma variável ponteiro.

int c = 10;

int\* ponteiro;

// ponteiro apontando para o mesmo endereco de c

ponteiro = &c;

// imprime o endereco da variavel c

printf("%d %d\n", ponteiro, &c);

// imprime o conteúdo da variavel c

printf("%d %d\n", \*ponteiro, c);

exercício proposto

Escreva uma função soma que recebe um ponteiro de inteiro num e mais dois inteiros a e b. A função deve calcular a soma de a+b em num.

A função ficará assim:

void soma(int\* num, int a, int b) {

\*num = a + b;

}

Sabendo disso, vamos então passar a variável tentativas como referência. Ou seja, vamos passar o endereço de memória dela para a função chuta, usando o &:

chuta(chutes, &tentativas);

Claro que se passamos um ponteiro de inteiro para a função, é porque ela recebe um ponteiro de inteiro. Precisamos mudar a assinatura do método chuta(), e fazê-lo receber um int\* tentativas. Além disso, precisamos usar o \* para acessar o conteúdo que o ponteiro tentativas aponta. Veja:

void chuta(char chutes[], int\* tentativas) {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[\*tentativas] = chute;

(\*tentativas)++;

}

Podemos agora colocar um printf em nosso jogo, mostrando o número de tentativas. Você verá que, como estamos passando o ponteiro da variável, o número é modificado na variável que queremos.

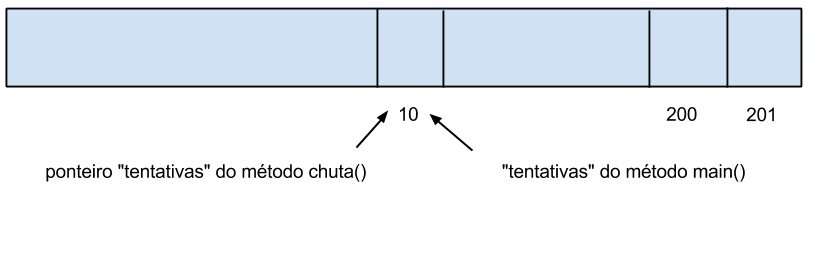
do {

printf("Você já deu %d chutes\n", tentativas);

// ...

}

Ponteiros, apesar de parecerem complicados em um primeiro instante, são poderosíssimos.



\*Está perdido? O código até o momento pode ser encontrado [aqui](https://gist.github.com/mauricioaniche/43137829f5ba0c3f9480).

Você deve estar se perguntando o motivo pela qual tivemos que passar o inteiro como ponteiro, mas não o array. Isso não muda: se quisermos mudar uma variável que está declarada em outro escopo, não tem solução: precisamos mexer direto em seu endereço de memória, e usar ponteiros.

E, acredite ou não, fizemos isso no array. Um array é por padrão um ponteiro. A variável chutes é um ponteiro que aponta para o inteiro que está na primeira posição do array. Sabemos que a primeira posição do array é chutes[0]. Se fizermos &chutes[0], temos o endereço de memória desse primeiro inteiro. Esse número é igual a chutes.

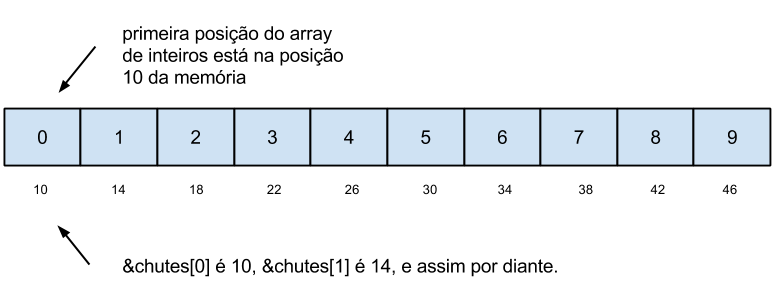
Veja o código:

printf("%d %d", chutes, &chutes[0]);

Agora que você está entendendo mais sobre a memória, veja que um array é uma "sequência de variáveis" declaradas. Em particular, essas variáveis estão exatamente uma do lado da outra. Quando declaramos um array de 10 posições, a máquina procura por um lugar onde caibam 10 inteiros, um ao lado do outro.

Quando fazemos chutes[2], por exemplo, a máquina então pega a posição de memória da primeira posição desse array, e vai navegando até a posição que quer. No exemplo, o número 2 quer dizer a terceira posição do array. Ele pula então 3 casas e chega no próximo inteiro.

Mas você lembra que discutimos o tamanho de um inteiro? Um inteiro ocupa 4 bytes. Isso quer dizer que se você declara uma variável inteira, a máquina separa 4 bytes da memória pra ele. Em um array, a mesma coisa, se temos um array de inteiros de 10 posições, precisamos de 40 bytes. Ou seja, se &chutes[0] nos retorna 10, &chutes[1] nos retornará 14, pois é onde o segundo inteiro começa.



Veja, em código. O programa abaixo declara um array de inteiros de 3 posições. Em seguida, imprimimos a posição de memória que estão guardados cada um desses números. Sabemos que um inteiro tem 4 bytes, e que os ítens de um array são guardados um ao lado do outro. Portanto, os números devem crescer de 4 em 4.

#include <stdio.h>

int main() {

int numeros[3];

printf("%d %d %d\n", &numeros[0], &numeros[1], &numeros[2]);

}

A saída (que é diferente a cada vez que executarmos o trecho de código, pois o endereço de memória muda) com os endereços, pulando de 4 em 4 bytes, como esperado:

1458119804 1458119808 1458119812

Podemos tratar esse int[] como um int\*. Afinal, arrays são ponteiros. Se temos um ponteiro de inteiro, e sabemos que ele aponta para um array, podemos acessar a primeira posição fazendo \*ponteiro. Se quisermos acessar a segunda posição, basta somarmos 1 a esse ponteiro, \*(ponteiro + 1). E assim por diante.

Veja o código abaixo, onde fazemos um loop, e acessamos os valores do array usando o tipo array convencional e usando diretamente o ponteiro:

#include <stdio.h>

int main() {

int numeros[3];

// declarando um ponteiro que aponta pro

// mesmo lugar que o array/ponteiro numeros

int\* ponteiro = numeros;

numeros[0] = 10;

numeros[1] = 20;

numeros[2] = 30;

for(int i = 0; i < 3; i++) {

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

e

// repare na soma que fazemos com o ponteiro

printf("%d ", \*(ponteiro + i));

printf("\n");

}

}

Sua cabeça a partir de agora deve estar muito diferente. Você consegue ver ponteiros em tudo? Lembre-se que variáveis são ponteiros para endereços de memória. A linguagem de programação geralmente nos esconde isso, e ela que se vira em manipular esse endereço e salvar nossos dados lá. Mas, se precisarmos, a linguagem C nos deixa lidar diretamente com esses endereços de memória.

Exercício proposto:

Escreva a mesma função de potencia que você fez na aula anterior, só que dessa vez, o resultado deve ser salvo em um inteiro que vem na lista de parâmetros da função. Para isso, claro, você precisará receber um ponteiro de inteiro:

int resultado;

potencia(&resultado, 10, 5);

Resposta:

Inicializamos uma variável resultado com 1, e aí iteramos b vezes em um for, cujo código é multiplicar o número atual por a.

void potencia(int\* resultado, int a, int b) {

\*resultado = 1;

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

\*resultado = \*resultado \* a;

}

}

A próxima função que vamos extrair de nosso código é a função que será responsável por nos dizer se o jogador já chutou determinada letra. Veja que é isso que o for de dentro faz. Ele varre a palavra secreta e a lista de chutes, e guarda na variável achou se o usuário já chutou ou não aquela letra.

É fácil ver que esse for usa o array de chutes, afinal, ele precisa passear por todo ele, e precisa também da letra que será procurada. Vamos escrever a função void jachutou(), que receberá um char letra, um char\* chutes (lembre-se que um array é um ponteiro, então podemos receber como um ponteiro), e a quantidade de tentativas.

Vamos também relembrar a lógica dessa função. Ela varria o array, procurando por um chute que seja igual a letra que ele quer comparar. Se ele achar a letra no array, então ele marcava a variável achou como 1. O 1, para nós, significa "verdadeiro". Caso contrário, achou era falso. Essa variável então era usada mais pra baixo, para exibir ou não a letra, mas isso não vem ao caso, já que não faz parte de dizer se uma letra já foi ou não chutada.

void jachutou(char letra, char\* chutes, int tentativas) {

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

}

Com a função escrita, vamos fazer uso dela e invocá-la no lugar onde o código antigo existia. O código ficou simples. Declaramos a variável achou, a inicializamos com 0, e invocamos nossa função jachutou. Em seguida, o if que imprime ou não a letra usa essa variável:

for(int i = 0; i < strlen(palavrasecreta); i++) {

int achou = 0;

jachutou(palavrasecreta[i], chutes, tentativas);

if(achou) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

Mas esse código ainda não funciona. Afinal, já sabemos que a variável achou, declarada dentro da função main() não é a mesma da declarada dentro da jachutou. Se quisermos "compartilhar" a variável, precisaremos passar o ponteiro dela. Sabemos que isso não é difícil, pois bastaria passarmos o endereço de achou, usando &achou, e receber o ponteiro do outro lado, como parâmetro, um int\* achou.

Apesar da alternativa ser válida, geralmente temos uma solução mais elegante pra isso. Veja que a variável achou é o "resultado" da nossa função jaachou. Ou seja, ela processa, faz um monte de coisa, mas ao final, precisa apenas "contar para quem chamou ela" o resultado dessa variável. Quando uma função precisa devolver apenas um dado (um número, ou char, ou double, ou qualquer outra coisa) para quem a chamou, damos um tipo de retorno para ela.

Até agora, sempre que declaramos uma função, usamos a palavra void ao começo. Usamos void quando a função não precisa retornar nada a quem chamou. Mas, se ela precisar retornar, podemos trocar essa palavra pelo tipo de retorno. Por exemplo, queremos devolver um inteiro, então trocamos a palavra void por int:

int jachutou(char letra, char\* chutes, int tentativas) {

// ...

}

Agora, precisamos indicar qual o valor que deve ser retornado por essa função. Para fazer isso, basta usarmos a palavra return. Quando essa palavra aparece dentro de uma função, geralmente acompanhada por um valor (por exemplo, return 10; para retornar o valor 10), a função para ali e quem chamou essa função recebe esse valor.

Aqui, queremos retornar o conteúdo da variável achou ao final da função. Então colocaremos como última linha, esse retorno. Além disso, precisamos declarar a variável achou no começo dessa função e inicializá-la com 0, pois ela não existia ali até então:

int jachutou(char letra, char\* chutes, int tentativas) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

Agora, nossa função jachutou está pronta. Ela nos devolve um inteiro sempre que invocarmos ela. Esse inteiro vale 1 quando ele achar a letra, e 0 quando não achar. Veja que isso parece uma função da matemática. Dado um conjunto de valores de entrada, ela nos devolve um resultado como saída. É por isso que chamamos de função.

Como matemática nos dá bons exemplos de funções, poderíamos ter uma função que nos devolva o quadrado de um número. Ela receberia um número como parâmetro e devolveria a multiplicação dele por ele mesmo. Veja, em código:

int quadrado(int n) {

return n \* n;

}

Usar essa função é igual igual usar funções que devolvem void. A diferença, claro, é que elas nos retornam algo. Para capturarmos o valor retorno delas, precisamos guardá-la em uma variável. Veja, por exemplo, alguns exemplos de uso da função quadrado(), algumas delas guardando em variáveis, outras usando-as diretamente:

// guardando em uma variável

int resultado = quadrado(2);

printf("%d", resultado);

// usando diretamente em um if

if(quadrado(3) < 10) {

printf("Já sabia, o quadrado de 3 é 9");

}

// usando direto em um printf

printf("%d", quadrado(5));

Voltando ao jogo, vamos agora fazer uso direito da função jachutou. Como vimos que podemos usá-la diretamente dentro do if, sem a necessidade de uma variável auxiliar, vamos usá-la diretamente no condicional que decide mostrar ou não a letra. Veja como nosso código está cada vez mais simples:

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i], chutes, tentativas)) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

Escreva uma função soma que recebe um array de inteiros e o tamanho do array, e retorna a soma dos números desse array.

int nums[3];

nums[0] = 10;

nums[1] = 20;

nums[2] = 30;

int total = soma(nums, 3);

Repare no código acima que arrays são por natureza ponteiros, então podemos passá-los diretamente, sem o uso de &.

Resposta proposta:

Repare que recebemos o ponteiro e varremos ele inteiro, somando.

int soma(int\* nums, int tam) {

int total = 0;

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

total += nums[i];

}

return total;

}

Está perdido? O código até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/8f4365b14086136ce2fa).

Vamos continuar separando em funções. Agora, podemos extrair o loop for de fora, aquele que varre a palavra secreta, para uma função desenhaforca(). Extraí-la é fácil. Basta fazermos a função receber a palavra secreta, os chutes, e a quantidade de tentativas. Como ela não precisa retornar nada, ela é void:

void desenhaforca(char\* palavrasecreta, char\* chutes, int tentativas) {

printf("Você já deu %d chutes\n", tentativas);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i], chutes, tentativas)) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

Fazendo uso dessa função no lugar certo, o loop principal do jogo ficou mais mais simples de ser lido e entendido. Dá até pra ler em português agora o que acontece. Ele "desenha forca" e "chuta" enquanto "não acertou" e "não enforcou". Veja que você consegue entender o algoritmo em alto nível em poucos segundos. O desenvolvedor só precisará ler os detalhes do algoritmo (ou seja, a implementação de cada um desses métodos) se quiser ou tiver necessidade.

Nosso código está cada vez mais bem escrito e isolado, e isso é fundamental quando programamos.

do {

desenhaforca(palavrasecreta, chutes, tentativas);

chuta(chutes, &tentativas);

} while (!acertou && !enforcou);

Por fim, ainda podemos deixar mais fácil de ser entendido a parte onde selecionamos a palavra secreta. Por enquanto, ela é fixa, mas mais pra frente vamos ler de um arquivo de palavras. Se já tivermos esse código bem isolado em uma função, ficará fácil depois trocar. A função escolhepalavra() receberá o array palavrasecreta:

void escolhepalavra(char\* palavrasecreta) {

sprintf(palavrasecreta, "MELANCIA");

}

Vamos invocá-la também em um lugar mais oportuno, de melhor legibilidade, um pouco mais afastado da declaração das variáveis. Logo abaixo da invocação da função abertura(), por exemplo, parece bom. A função main agora está bastante enxuta:

abertura();

escolhepalavra(palavrasecreta);

do {

// ...

} while (...);

Repare que todas nossas funções agora precisam passar os mesmos parâmetros pra lá e pra cá. E faz sentido, afinal o array de chutes, a palavra secreta e o número de tentativas, são variáveis importantes para o jogo. Esse problema, inclusive, nos serviu de motivação para nossa discussão de funções, parâmetros e ponteiros.

E se conseguíssemos declarar uma variável que fosse visível para mais de uma função, e não só naquela na qual ela foi declarada? Nosso código ficaria bem menor, afinal, a **assinatura da função** (assinatura é o nome que damos para o nome, parâmetros e tipos de retorno de uma função) seria bem mais simples.

Na verdade, podemos. Em C, podemos declarar uma variável fora do arquivo. Essas variáveis são chamadas de **variáveis globais**, pois elas ficam visíveis e passíveis de serem acessadas por qualquer função declarada em nosso programa. Elas são bastante úteis quando temos variáveis que são importantes, e por consequência, manipuladas por todas as nossas funções.

Vamos fazer isso então com as variáveis chutes, palavrasecreta e tentativas. Vamos declará-las logo após a lista de includes:

#include <stdio.h>

#include <string.h>

char palavrasecreta[20];

char chutes[26];

int tentativas = 0;

E, já que agora, elas são visíveis para todas as funções, não as precisamos mais passar por parâmetro em todas as funções. Vamos então remover os parâmetros que não são mais necessários em todas as funções que temos.

Na função chuta(), não precisamos mais nem de chutes nem de tentativas como parâmetro. E já que tentativas agora é um inteiro convencional (e não mais um ponteiro), vamos tirar as operações de ponteiro e fazer operações comuns com ela:

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

}

A função jachutou() também não precisa mais do array de chutes, nem da variável tentativas. Aqui a mudança é simples, e basta removermos da assinatura da função. Repare que ainda assim precisamos receber a letra, que procuraremos no array. Veja como ela ficou mais simples:

int jachutou(char letra) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

A função desenhaforca() também não precisa receber mais nenhum parâmetro, e nem passar todos eles para a função jachutou(), invocada dentro dela.

void desenhaforca() {

printf("Você já deu %d chutes\n", tentativas);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i])) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

O mesmo acontece com a função escolhepalavra():

void escolhepalavra() {

sprintf(palavrasecreta, "MELANCIA");

}

Por fim, a main também não precisa mais passar os parâmetros para as funções. Veja só como ela ficou totalmente enxuta e fácil de ser lida:

int main() {

int acertou = 0;

int enforcou = 0;

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

} while (!acertou && !enforcou);

}

Veja então que agora escolhemos o melhor lugar para declarar nossas variáveis. Variáveis que fazem sentido existirem em um contexto mais amplo, e são diretamente relacionadas com o nosso jogo de forca, como palavrasecreta, chutes e tentativas são globais, e visíveis por todas as funções. Variáveis que são mais locais, específicas de uma função em particular, como é o caso da variável achou na função jachutou, ou mesmo a variável i na função desenhaforca, são declaradas locais e existem só naquele contexto.

Será então que variáveis globais são sempre boas? Algo para você pensar desde o começo de sua carreira em desenvolvimento de software é que toda decisão de código tem sempre prós e contras. Sem exceção. Ainda não fomos capazes de criar balas de pratas, ou seja, soluções perfeitas e exatas para todos os problemas. Variáveis globais nos ajudam a diminuir a quantidade de código que escrevemos, afinal, precisamos passar menos parâmetros pra lá e pra cá, e as funções já as acessam diretamente.

A desvantagem é que agora fica mais difícil entender quem modificou o quê. Antes, quando as variáveis estejam sempre declaradas dentro de escopos controlados, se você quisesse saber quem mudou a variável tentativas, bastava olhar a função em que ela estava declarada. Agora, você precisa olhar o código todo. Ou seja, perdemos um pouco do controle sobre o acesso às variáveis, já que qualquer função pode alterá-la. É uma troca.

Variáveis globais são geralmente muito criticadas por isso. Mas se você as usar bem, não terá problemas. Lembre-se de colocar como global, apenas aquelas variáveis que tenham um sentido maior para o código. Mais pra frente (e no mundo real), criaremos programas com vários arquivos. Esse é o real problema: ter muitos arquivos diferentes, que fazem coisas diferentes, mexendo nas mesmas variáveis. Imagine o código do seu jogo de adivinhação mexendo nas variáveis do jogo de forca? Não faz sentido. Mesmo que inventássemos alguma regra em que os jogos se combinassem, o ideal seria que só o arquivo forca.c mexesse em variáveis do jogo de forca, e o adivinhacao.c mexesse em variáveis do jogo de adivinhação.

Neste curso, não falamos sobre orientação a objetos, mas sem dúvida, é o próximo assunto que você deve aprender. **Orientação a objetos** é uma outra maneira de escrever programas de computador, que pensa diferente da **programação procedural**, que é o que estamos usando aqui, com a linguagem C, onde temos um conjunto de funções que trabalham juntas. Existem muitas maneiras diferentes de se escrever um programa de computador, e em todas elas você deve pensar em facilitar a manutenção do seu código. Como fazer isso? Deixando sempre perto as coisas que mudam juntas. Tudo que é relacionado ao forca, fica em um arquivo, e tudo que é relacionado ao adivinhação, fica em outro arquivo.

Nosso jogo agora, além de bonito por fora, está ficando bonito por dentro.

É hora de continuarmos nossa forca. Chegue no mesmo ponto que o vídeo.

O código ficará assim:

#include <stdio.h>

#include <string.h>

char palavrasecreta[20];

char chutes[26];

int tentativas = 0;

void abertura() {

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

}

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[tentativas] = chute;

tentativas++;

}

int jachutou(char letra) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

void desenhaforca() {

printf("Você já deu %d chutes\n", tentativas);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i])) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

void escolhepalavra() {

sprintf(palavrasecreta, "MELANCIA");

}

int main() {

int acertou = 0;

int enforcou = 0;

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

} while (!acertou && !enforcou);

}

Falta pouco pra terminarmos nosso jogo. Precisamos, nesse momento, fazer toda a lógica de vencer ou perder o jogo. Em um jogo de forca, perdemos quando perdemos todas nossas chances. Ou ganhamos quando acertamos a palavra secreta. Mas, agora que já aprendemos a criar funções, tudo fica mais fácil. Antes, precisávamos olhar aquelas centenas de linhas de código e achar o ponto exato. Agora, podemos escrever o código em funções isoladas, e depois apenas invocá-las.

Vamos começar escrevendo a função que nos diz que o jogador perdeu o jogo. Basta contarmos o número de chutes incorretos que ele deu. Se esse número for maior que o número de tentativas permitidas, ele perdeu. Nossa função, denominada enforcou() nos devolverá então 1 se ele foi enforcado, ou 0 se ele ainda não foi. O número de tentativas será 5 (cabeça, corpo, braços, pernas e fim).

A lógica dessa função será a seguinte: declararemos inicialmente uma variável chamada erros, que contará o número de letras que o usuário chutou de maneira incorreta. Essa variável nos será útil ao final, pois, se ela for maior-ou-igual a 5, então o jogador deve ser enforcado. Então, faremos um loop na lista de chutes e, para cada chute dado, verificaremos se a letra existe na palavra secreta. Se existir, quebramos o loop, e passamos a frente. Aqui, usaremos a mesma estrategia que estamos usando quando temos loops encadeados. A variável existe nos ajudará a saber se a letra foi encontrada no loop de dentro. Caso contrário, somaremos 1 à variável erros, e continuamos.

Veja, em código:

int enforcou() {

int erros = 0;

// vamos fazer o loop em todos os chutes dados

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

int existe = 0;

// agora vamos olhar letra a letra da palavra secreta

// e ver se encontramos o chute aqui

for(int j = 0; j < strlen(palavrasecreta); j++) {

if(chutes[i] == palavrasecreta[j]) {

// encontramos, vamos quebrar o loop

existe = 1;

break;

}

}

// se nao encontrou, soma um na quantidade de erros

if(!existe) erros++;

}

// se tivermos mais do que 5 erros, retornamos 1

// caso contrario, retornamos 0.

return erros >= 5;

}

Repare no return em uma linha. Acostume-se com códigos assim. Lembre-se que a máquina vai primeiro avaliar o elemento da direita, ou seja, a expressão condicional e depois fazer o retorno. Ou seja, se erros for maior que 5, ela avaliará isso para 1 (verdadeiro). Caso contrário, 0. E a função retornará o resultado disso.

Precisamos agora invocar essa função. Como ela está bem isolada, podemos facilmente tocar a variável enforcou, que havíamos declarado la na nossa main (mas que nunca usamos), pela invocação da função. Precisamos, claro, deletar a variável:

int main() {

int acertou = 0;

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

// aqui invocamos a função enforcou

} while (!acertou && !enforcou());

}

Se você rodar o programa agora, e fizer 5 tentativas erradas, verá que nosso programa parará.

Conforme vamos desenvolvendo nosso software, e aprendendo com ele, mudamos de ideia. Por exemplo, a variável tentativas, cuja ideia era contar a quantidade de chutes dados, agora ficou ambígua. Tentativas pode ser o número de chances que ele tem. Se um novato pegar esse código, ele precisará ler mais do código pra entender.

Vamos então **refatorar** nosso código. Refatorar é mudar o código, geralmente para melhor, mas sem mudar o que ele faz. Ou seja, deixá-lo mais bonito e mais fácil de ser mantido. Uma refatoração comum é, por exemplo, mudar o nome das coisas, como variáveis, métodos ou até mesmo arquivos.

Trocaremos o termo tentativas por chutesdados, que deixa bem mais claro o que ela guarda lá. Precisamos trocar na declaração e, óbvio, em todos os lugares que a usamos:

int chutesdados = 0;

A próxima etapa é descobrir se ganhamos ou não o jogo. A abordagem será a mesma, mas "ao contrário". Agora, passaremos primeiro pela palavra secreta. Para cada letra, descobriremos se o usuário já chutou aquela letra. Caso alguma letra não tenha sido chutada, podemos matar logo a função e retornar falso. Ao final, se ele chutou todas as letras que existem na palavra, ele ganhou o jogo.

Essa função é bem mais simples, afinal já temos a função jachutou que nos diz se uma letra já foi chutada ou não. Então, vamos apenas passear pela palavra secreta e ver se a letra já foi chutada. Se ela não foi chutada (então negaremos o if), já fazemos um return 0. Repare que podemos ter returns a qualquer momento do nosso código; a máquina parará a função naquele momento, e retornará o resultado direto. Vamos declarar essa função logo abaixo da jachutou():

int ganhou() {

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(!jachutou(palavrasecreta[i])) {

return 0;

}

}

return 1;

}

Para vê-la funcionar, basta também colocarmos sua invocação dentro do do-while e remover a inútil variável acertou. Pronto, nosso jogo agora para se o usuário ganhou ou perdeu:

int main() {

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

} while (!ganhou() && !enforcou());

}

Pronto. Nosso jogo agora tem começo, meio e fim. Só não está bonito, mas mais tarde o embelezamos.

### Erro de compilação?

Seu código não está compilando? Pode acontecer. Veja o próximo vídeo, que resolveremos o seu possível problema.

Está perdido? O código implementado até o momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/a4eb153f5da8f2732e42).

É provável que nesse momento, o seu código compile ou não. Repare que em nenhum momento eu comentei onde devemos colocar cada função que declaramos. Será que existe uma ordem específica? Sim, existe. O compilador da linguagem C não é tão esperto, e a ordem da declaração das funções é importante. Quando o compilador está passando pelo arquivo, se ele encontra uma invocação de função que ele não conhece, ele opta por reclamar, ao invés de continuar e procurar pela declaração mais a frente (isso é feito por compiladores de linguagens mais modernas). O erro que você deve ter tomado é parecido com o abaixo:

forca.c:31:7: warning: implicit declaration of function 'jachutou'

is invalid in C99 [-Wimplicit-function-declaration]

Mas, existe uma maneira de resolver isso. É declarar todas as assinaturas de funções que aparecerão naquele arquivo, antes de darmos o código delas. A assinatura de uma função é aquela primeira linha que usamos para declará-la, que contém o tipo de retorno, o nome e os parâmetros que ela recebe.

Vamos colocá-las todas, uma-a-uma, logo após os includes. Aqui optamos por colocar antes das variáveis globais, mas não há diferença.

#include <stdio.h>

#include <string.h>

// lista de funções que aparecerão no arquivo

int enforcou();

void abertura();

void chuta();

int jachutou(char letra);

int ganhou();

void desenhaforca();

void escolhepalavra();

// variáveis globais

char palavrasecreta[20];

char chutes[26];

int chutesdados = 0;

// agora aqui começam as implementações das funções,

// do jeito que você está acostumado.

Repare que agora você pode ter as suas funções em qualquer ordem. Coloque a função ganhou() antes da jachutou(), por exemplo, e veja que o código compila normalmente.

Esse é um problema comum de programas escritos em C. E esse problema é ainda mais grave quando temos nosso código espalhados em arquivos diferentes, e queremos usar funções de um arquivo em outro. Por esse motivo, apesar do nosso código funcionar com a declaração das funções acima, temos o costume de colocá-la isso em um arquivo separado. Esse arquivo é conhecido como **header file** e tem a extensão .h.

Veja que você já viu essa extensão antes. É a mesma extensão do stdlib, string e todos os outros que importamos. Ou seja, em algum lugar, existe um stdlib.c que contém a implementação das funções, e como as queremos usar em nosso programa, importamos a declaração delas, que está todas no .h.

Vamos criar então o arquivo forca.h e colocar essas declarações lá:

// lista de funções que aparecerão no arquivo

int enforcou();

void abertura();

void chuta();

int jachutou(char letra);

int ganhou();

void desenhaforca();

void escolhepalavra();

Agora, em nosso forca.c, o importaremos. Repare que aqui usamos uma aspa ao invés dos sinais de maior e menor. Afinal, esse é um header file que é nosso e está no diretório corrente:

#include <stdio.h>

#include <string.h>

#include "forca.h"

// variáveis globais

char palavrasecreta[20];

char chutes[26];

int chutesdados = 0;

// agora aqui começam as implementações das funções,

// do jeito que você está acostumado.

Agora, sempre que criarmos uma nova função, precisamos declará-la também no nosso header file. Acostume-se com isso, pois em nosso próximo jogo, dividiremos melhor os arquivos, e usaremos funções pra lá e pra cá, e esses arquivos serão importantes.

A única parte chata do jogo é que a palavra secretá está fixa. Depois que ele adivinhar a primeira vez, acabou. Precisamos dar um jeito de fazer com que a palavra seja randômica. Criar números randômicos é fácil, mas agora palavras que façam sentido... isso é, com certeza, um problema computacional bastante desafiador.

O que faremos é escolher uma palavra aleatória de um arquivo de texto, que será nosso "banco de dados de palavras". Esse arquivo pode ter qualquer tamanho, ou seja, você poderá colocar novas palavras lá sempre que quiser, e o programa deve entender isso.

Vamos começar então escrevendo um arquivo de exemplo. Sempre que queremos criar um arquivo que será lido pelo computador, precisamos definir um formato. Como separaremos as palavras? Por vírgula? Por enter? O formato é importante, afinal, o programa precisará saber como interpretá-lo.

Nosso arquivo terá o seguinte formato: a primeira linha nos indicará quantas palavras existem naquele arquivo. Em seguida, teremos as várias palavras, uma em cada linha. Isso facilitará a nossa leitura via código. Veja o arquivo, que chamaremos de palavras.txt:

3

MELANCIA

MORANGO

MELAO

Agora vamos começar a ler esse arquivo. A primeira coisa que precisamos fazer é abrí-lo. Para isso, precisamos declarar um ponteiro do tipo FILE\*, e usar a função fopen() que abre um arquivo do disco. Essa função nos devolve então um ponteiro (que guardaremos na variável já declarada), e a partir daí, passamos esses ponteiros para as funções que lerão os caracteres desse arquivo. Repare no "r". Isso indica que estamos abrindo o arquivo somente para leitura (poderíamos tê-lo aberto para escrita).

Todo o código estará dentro da função escolhepalavra():

void escolhepalavra() {

FILE\* f;

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

}

Essa é a primeira vez que estamos lidando com recursos de I/O, ou seja, de entrada ou saída. Na prática, elas podem falhar, pois o arquivo pode não estar disponível, ou não termos permissão de leitura, etc. Então, sempre que lidamos com isso, precisamos tratar as possíveis falhas.

A função fopen() devolve um ponteiro com endereço 0, caso algum erro tenha acontecido. Vamos tratar esse possível erro, e caso ele tenha acontecido, devolveremos uma mensagem para o usuário e acabaremos com nosso programa ali mesmo (usando a função exit(), que finaliza a aplicação). Repare que o parâmetro 1, passado ao exit indica ao sistema operacional que o programa terminou de maneira que ele não gostaria; ou seja, um erro ocorreu. Se tivéssemos retornado 0, o sistema operacional saberia que o programa rodou de acordo com o esperado. A função exit() está declarada em stdlib.h. Precisamos colocar o include:

#include <stdlib.h>

void escolhepalavra() {

FILE\* f;

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

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

}

Agora sim, estamos prontos para começar a ler nosso arquivo, e sabemos que na primeira linha, temos a quantidade de palavras que leremos na sequência. Vamos capturar esse número. Você já sabe mais ou menos como ler do arquivo. Em C, temos a função fscanf(), similar a função que você já conhece. A diferença dela é que o primeiro parâmetro é o ponteiro para um arquivo.

Se quisermos ler um inteiro de um arquivo, por exemplo, fazemos igual ao código abaixo. Repare que a função sabe que o número "acabou" quando encontrar um enter no arquivo. Ou seja, nesse caso, ela lê até achar um fim de linha.

int qtddepalavras;

fscanf(f, "%d", &qtddepalavras);

Toda vez que lemos de um arquivo, a leitura é sequencial. Ou seja, se já capturamos o primeiro número, a próxima vez que chamarmos a função fscanf(), ela continuará a leitura do ponto que parou. É como se ela tivesse uma pequena seta, que aponta para a posição que está lendo no arquivo no momento.

Vamos ler então a próxima coisa que aparece no arquivo. Sabemos que é uma palavra. Vamos declarar um array e ler, usando a máscara "%s". Dado que em nosso exemplo, sabemos que temos 3 palavras lá, vamos ler 3 palavras de uma só vez, apenas para teste:

char palavra1[50];

fscanf(f, "%s", palavra1);

char palavra2[50];

fscanf(f, "%s", palavra2);

char palavra3[50];

fscanf(f, "%s", palavra3);

Nesse momento, se imprimirmos as 3 strings, teremos ::MELANCIA, MORANGO:: e ::MELAO::, igual esperávamos. Mas, nesse nosso jogo em particular, não precisamos salvar todas elas, mas sim somente uma (a que escolhermos de maneira aleatória). Vamos então gerar um número aleatório, de 0 até a quantidade de elementos, e salvar apenas a palavra que estiver naquela posição.

Não temos como pular direto para a linha que queremos, então precisaremos fazer um loop com a função fscanf() e, na linha certa, salvar o retorno. Para facilitar, vamos salvar a palavra secreta já na variável correta palavrasecreta. Repare que a última interação do loop será justamente a linha randômica escolhida. O loop acabará e a avariável terá a palavra secreta que queremos:

int qtddepalavras;

fscanf(f, "%d", &qtddepalavras);

// gera numero aleatorio

// nao esqueça de incluir time.h

srand(time(0));

int randomico = rand() % qtddepalavras;

// lê do arquivo até chegar na linha desejada

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

fscanf(f, "%s", palavrasecreta);

}

Nosso código está quase pronto. Já selecionamos uma palavra aleatória dentro do nosso banco de dados de palavras. Só estamos esquecendo uma coisa: sempre que lidamos com I/O, precisamos lembrar de abrir o arquivo (ou qualquer outra fonte de entrada), usá-la e, ao final, fechá-la.

Fechar é importante. Ele libera aquele recurso para que outros programas consigam fazer uso dele também. Quando abrimos um arquivo para leitura, o sistema operacional pode bloquear a leitura dele de outros aplicativos até que o programa corrente termine seu trabalho. Ou seja, feche o mais rápido possível. Fazemos isso com a função fclose().

O método escolhepalavra() completo ficou:

void escolhepalavra() {

FILE\* f;

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

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtddepalavras;

fscanf(f, "%d", &qtddepalavras);

srand(time(0));

int randomico = rand() % qtddepalavras;

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

fscanf(f, "%s", palavrasecreta);

}

fclose(f);

}

Veja só que ao longo dessa seção, só alteramos um ponto do nosso arquivo, que foi a função escolhepalavra(). E tudo funciona. Veja só como dividir nosso programa em funções bem definidas é realmente útil. Podemos fazer mudanças em pequenas partes do programa sem nos preocuparmos com o todo. É isso que faz um código ser fácil de ser mantido: quando conseguimos mexer apenas em pontos específicos, sem a necessidade de ler ou alterar todo o código.

Está perdido? O código implementado até esse momento pode ser encontrado [*aqui*](https://gist.github.com/mauricioaniche/dd0b366ad9c4f7ee0cd6).

A próxima funcionalidade em nosso jogo é perguntar ao usuário se ele deseja adicionar uma nova palavra no banco de dados. Como ela funcionará? Ao final do jogo, perguntaremos se ele deseja fazer isso. Se ele nos responder 'S' (sim), então, perguntaremos a palavra e o adicionaremos no arquivo.

Vamos começar pela parte fácil, que é perguntar se o usuário quer ou não adicionar uma palavra. E isso, claro, em uma nova função, isolada do resto do nosso programa. Lembre-se que, como vamos capturar um único char, precisamos colocar um espaço na máscara, para que o scanf() ignore o enter ao final:

void adicionapalavra() {

char quer;

printf("Você deseja adicionar uma nova palavra no jogo (S/N)?");

scanf(" %c", &quer);

}

Se ele responder "S", então perguntaremos a ele a palavra. Também não há novidade:

void adicionapalavra() {

char quer;

printf("Você deseja adicionar uma nova palavra no jogo (S/N)?");

scanf(" %c", &quer);

if(quer == 'S') {

char novapalavra[20];

printf("Digite a nova palavra, em letras maiúsculas: ");

scanf("%s", novapalavra);

// agora falta salvar no arquivo

}

}

Agora precisamos salvar no nosso arquivo de banco de dados. Para abrir um arquivo, precisamos fazer uso da função fopen() que já conhecemos. O que mudará são os parâmetros que passaremos para ela. O r indicava leitura. Usaremos aqui o a, que significa, "append", ou seja, que vamos inserir no fim do arquivo.

Com um FILE aberto, tudo o que precisamos fazer é usar a função fprintf(), análoga a printf(), que você já conhece. A diferença é que o primeiro parâmetro que ela recebe é justamente o arquivo em que ela escreverá (você percebeu que o f, tanto da fprintf quanto da fscanf é de "file?").

Também não esqueceremos de tratar um possível erro na abertura do arquivo. Ele pode não ter acontecido no começo do jogo, mas pode acontecer ao final. Em código:

if(quer == 'S') {

char novapalavra[20];

printf("Digite a nova palavra, em letras maiúsculas: ");

scanf("%s", novapalavra);

FILE\* f;

// abre arquivo

f = fopen("palavras.txt", "a");

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

// escreve a palavra nele

fprintf(f, "%s", novapalavra);

// fecha

fclose(f);

}

Até o momento então, aprendemos que o ponteiro FILE\* aponta para um arquivo, e passamos esse ponteiro para todas as outras funções que o manipulam. O fscanf() consegue ler dados de um arquivo e o fprintf consegue escrever nele. Mas, tudo depende da configuração passada para a função fopen(): se queremos apenas ler, usamos "r", e se queremos "appendar", ou escrever ao final, usamos "a".

### Não esqueça de fechar o arquivo

Fechar o arquivo com o fclose() é especialmente importante em momentos de escrita. Isso porquê muitas vezes a máquina opta por não escrever no disco no momento em que você invoca um fprintf(); ela prefere "cachear", ou seja, guardar a modificação em memória e só salvar no disco depois. Ela toma essa decisão porque ficar indo ao disco e escrevendo fisicamente nele, de pouco em pouco, pode ser uma operação demorada. Ela prefere, então, esperar mais texto e ir de uma só vez ao disco rígido. A operação de fechar, portanto, diz à máquina que você acabou tudo o que tinha que escrever e que agora, independente do tamanho desse buffer, ela precisa escrever no disco. Se o programador, por descuido, esquece de fechar o arquivo, a máquina não escreverá no arquivo, e ele terá a sensação de que seu código não funcionou.

Portanto, feche sempre o arquivo que você abriu.

Mas veja que somente escrever ao final não nos é suficiente. Nosso arquivo de banco de dados tem um formato diferente, pois a primeira linha indica a quantidade de palavras. Ou seja, além de adicionar a palavra, precisamos ainda modificar o número que está no começo. Para isso, precisamos fazer uso de um outro modo de abertura de arquivo. O "w", por exemplo, nos ajuda a escrever em arquivos, mas ele só escrever em arquivos novos, ou seja, se o arquivo existir, ele o apaga primeiro.

O modo que precisamos aqui é o r+. Ele nos permite ler e alterar um arquivo. E é exatamente o que precisamos: o primeiro passo é ler o número que está no começo do arquivo, incrementá-lo e sobrescrevê-lo. Em seguida, adicionar a nova palavra ao final.

Vamos começar então abrindo o arquivo nesse novo modo e lendo o número que está na primeira linha:

f = fopen("palavras.txt", "r+");

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtd;

fscanf(f, "%d", &qtd);

Agora, precisamos incrementar essa variável e sobrescrevê-la no arquivo. Mas, como já lemos o inteiro, o "ponteiro" que aponta para o lugar do arquivo que a leitura continuará está mais a frente. Precisamos voltar esse ponteiro e posicioná-lo no lugar que desejamos começar a escrever. Para mudar o ponteiro de posição, usamos a função fseek(). Ela recebe três parâmetros: o arquivo, quantos bytes ela deve andar (para a esquerda ou para a direita) e da onde ela deve começar a andar (do começo do arquivo, da posição corrente, ou do final). Vamos posicioná-la então "0 bytes" a partir do começo do arquivo:

qtd++;

fseek(f, 0, SEEK\_SET);

Repare que SEEK\_SET é uma constante. Poderíamos usar também as SEEK\_CUR para andar a partir do ponto atual, ou mesmo SEEK\_END, para andar a partir do fim do arquivo. Com o ponteiro posicionado na posição que queremos, basta agora escrevermos a variável qtd, usando o fprintf(). A máquina não pensará duas vezes e escreverá por cima do que está naquela posição:

fprintf(f, "%d", qtd);

Agora, por fim, precisamos escrever a palavra no final do arquivo. Vamos novamente usar fseek() para ir ao fim do arquivo, e fprintf() para escrever a palavra. Repare o \n no começo da string. Precisamos disso, pois o ponteiro nos posiciona bem ao final do arquivo que, teoricamente, não tem um enter ao final. Então, escrevemos um enter e aí sim a palavra:

fseek(f, 0, SEEK\_END);

fprintf(f, "\n%s", novapalavra);

Pronto. Agora o jogador pode adicionar novas palavras ao final da partida:

void adicionapalavra() {

char quer;

printf("Você deseja adicionar uma nova palavra no jogo (S/N)?");

scanf(" %c", &quer);

if(quer == 'S') {

char novapalavra[20];

printf("Digite a nova palavra, em letras maiúsculas: ");

scanf("%s", novapalavra);

FILE\* f;

f = fopen("palavras.txt", "r+");

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtd;

fscanf(f, "%d", &qtd);

qtd++;

fseek(f, 0, SEEK\_SET);

fprintf(f, "%d", qtd);

fseek(f, 0, SEEK\_END);

fprintf(f, "\n%s", novapalavra);

fclose(f);

}

}

# Para saber mais: Mais sobre I/O

A biblioteca de entrada e saída da linguagem C é bastante extensa. Mesmo as funções estudadas aqui podem fazer muito mais do que o mostrado. Todas as funções, por exemplo, possuem retorno. A própria fscanf(), por exemplo, retorna o número de itens devolvidos. Por exemplo, se a máscara for "%d", ela retornará 1, se tudo der certo. Se a máscara for "%d %d", ou seja, capturar 2 inteiros, ela retornará 2.

int n1, n2;

int sucesso = fscanf("%d %d", &n1, &n2);

if(sucesso == 2) {

printf("Os dois números: %d e %d", n1, n2);

}

else {

printf("Aconteceu um erro");

}

Podemos verificar também se o ponteiro atual do arquivo está apontando para o fim. Para isso, basta usar a função feof(). Ele pode ser bastante útil quando queremos ler o arquivo inteiro até o fim. Por exemplo, se quisermos ler caractere a caractere do arquivo e imprimí-lo, precisamos repetir isso enquanto não chegarmos ao fim do arquivo.

Por enquanto, a única função de leitura que vimos até então é a fscanf(). Ela é importante quando queremos ler caracteres ASCII e strings. Mas, muitas vezes precisamos ler um byte, um char, ou mesmo uma struct (que aprenderemos no próximo jogo). Nesses casos, precisamos ler e escrever dados de maneira "mais bruta". A função fgetc(), por exemplo, lê apenas o próximo char.

FILE\* f;

char c;

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

while(!feof(f)) {

c = fgetc(f);

printf("Char %c\n", c);

}

Nesses casos também, as funções fread() e fwrite() podem ajudar. Elas lêem uma quantidade específica de bytes e as guardam na estrutura que você passar, seja ela um array, uma variável ou uma struct. No próximo jogo, faremos uso delas.

Perceba que daqui pra frente você começará a aprender e fazer uso de funções que são complexas e fazem muita coisa. A melhor maneira de entendê-las melhor é ler o manual oficial delas. Nisso, o Google te ajudará. Sempre que tiver dúvida dos parâmetros que uma função recebe, o que ela retorna e como interpretar esse retorno, procure pela sua definição na internet. Buscar é também parte do dia a dia do programador.

Continue agora o jogo, chegando no mesmo ponto em que chegamos no vídeo.

O código ficará assim:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include "forca.h"

char palavrasecreta[20];

char chutes[26];

int chutesdados = 0;

int enforcou() {

int erros = 0;

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

int existe = 0;

for(int j = 0; j < strlen(palavrasecreta); j++) {

if(chutes[i] == palavrasecreta[j]) {

existe = 1;

break;

}

}

if(!existe) erros++;

}

return erros >= 5;

}

int ganhou() {

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(!jachutou(palavrasecreta[i])) {

return 0;

}

}

return 1;

}

void abertura() {

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

}

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

chutes[chutesdados] = chute;

chutesdados++;

}

int jachutou(char letra) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

void desenhaforca() {

printf("Você já deu %d chutes\n", chutesdados);

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i])) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

void escolhepalavra() {

FILE\* f;

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

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtddepalavras;

fscanf(f, "%d", &qtddepalavras);

srand(time(0));

int randomico = rand() % qtddepalavras;

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

fscanf(f, "%s", palavrasecreta);

}

fclose(f);

}

void adicionapalavra() {

char quer;

printf("Você deseja adicionar uma nova palavra no jogo (S/N)?");

scanf(" %c", &quer);

if(quer == 'S') {

char novapalavra[20];

printf("Digite a nova palavra, em letras maiúsculas: ");

scanf("%s", novapalavra);

FILE\* f;

f = fopen("palavras.txt", "r+");

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtd;

fscanf(f, "%d", &qtd);

qtd++;

fseek(f, 0, SEEK\_SET);

fprintf(f, "%d", qtd);

fseek(f, 0, SEEK\_END);

fprintf(f, "\n%s", novapalavra);

fclose(f);

}

}

int main() {

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

} while (!ganhou() && !enforcou());

adicionapalavra();

}

Repetição de código é sempre problemático. Se temos o mesmo pedaço de código em muitos pontos da aplicação, isso implica em alterar todos os pontos quando necessário. Infelizmente, na prática, muitas vezes não propagamos a alteração para todos os lugares necessários, e isso nos traz bugs.

Você já tem ferramentas para evitar esse tipo de problema. Se você tem um trecho de código que é comum para várias partes do seu programa, basta extraí-lo e colocá-lo em uma função. Funções são facilmente reutilizáveis. E se você alterá-la, todos os lugares que a invocam farão uso da nova função.

Também já vimos que quando temos algum "número mágico", ou seja, um número fixo, podemos colocá-lo em uma constante. A constante é facilmente reutilizável e possui um nome semântico, muito melhor do que deixar o número jogado. E, se alterarmos a constante, a mudança propagará para todos os lugares que usam essa constante.

Nosso jogo de forca tem um número mágico, que já está espalhado pelo código. Veja que o tamanho máximo de uma palavra em nosso jogo é 20. Isso está definido na declaração do array palavrasecreta. E se quisermos mudar o tamanho da palavra, precisaremos mudar a declaração do array. Mas não é só isso: precisamos mudar também dentro da função adicionapalavra(), pois lá também declaramos um array com o mesmo tamanho. No momento, são apenas 2 pontos; mas amanhã serão 3, e depois de amanhã 4.

Vamos já corrigir isso, criando uma constante para isso. E dessa vez, a colocaremos no header file. Faremos um #define por lá, e o reutilizaremos em nosso código. Veja:

// forca.h

#define TAMANHO\_PALAVRA 20

// forca.c

char palavrasecreta[TAMANHO\_PALAVRA];

void adicionapalavra() {

// ...

char novapalavra[TAMANHO\_PALAVRA];

}

Até então, só mostramos o número de chutes dados pelo jogador, mas a parte mais legal do jogo de forca é justamente exibir o homenzinho enforcado. Vamos lá então, dentro da função desenhaforca() fazer um desenho mais amigável:

void desenhaforca() {

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | (\_) \n");

printf(" | \\|/ \n");

printf(" | | \n");

printf(" | / \\ \n");

printf(" | \n");

printf("\_|\_\_\_ \n");

printf("\n\n");

// ...

}

Mas não podemos exibir o boneco cheio o tempo inteiro. Isso depende do número de erros. Se o número de erros é 0, exibimos apenas a corda. Se ele errou 1 vez, exibimos a cabeça; 2 vezes, a cabeça e o corpo; 3 vezes, a cabeça, o corpo e os braços. 4 vezes, a cabeça, o corpo, os braços e as pernas.

Precisamos saber o número de vezes que o jogador já errou, então. Já temos algo parecido na função enforcou(). Ela calcula o número de erros e depois nos retorna se a quantidade de erros é maior que 5. Mas ela não nos serve exatamente porque precisamos saber a quantidade exata de erros. Ou seja, precisamos de um pedaço só dessa função.

Obviamente, não temos como utilizar parte de uma função. Mas podemos quebrar uma função em duas outras menores. Vamos dividir essa função enforcou() em duas: uma, nova, chuteserrados(), que calcula o número de chutes errados, e a outra, já existente, que fará uso da nova função e nos retornará verdadeiro se esse número for maior que 5:

// extraímos pra cá o pedaço daquela função

// que contava a quantidade de erros

int chuteserrados() {

int erros = 0;

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

int existe = 0;

for(int j = 0; j < strlen(palavrasecreta); j++) {

if(chutes[i] == palavrasecreta[j]) {

existe = 1;

break;

}

}

if(!existe) erros++;

}

return erros;

}

int enforcou() {

// usamos a função que acabamos de criar

return chuteserrados() >= 5;

}

Quebrar funções em funções menores é parte do nosso dia a dia. Fazemos isso para reutilizar pedaços de funções já existentes, ou mesmo para aumentar a legibilidade. Veja que a função enforcou() agora tem uma linha de código e é bem fácil de entender o que ela faz: se ele errou mais de 5 chutes, retorna falso. Qualquer desenvolvedor gasta muito pouco tempo para entendê-la.

Vamos aproveitar e extrair mais uma função. Repare que dentro da função chuteserrados() temos um loop ali que nos diz se uma letra existe ou não na palavra secreta. Para a função como um todo, isso é usado para contabilizar a quantidade de chutes errados. Mas só o algoritmo que nos diz se uma letra existe na palavra nos será útil mais pra frente. Extraindo, temos:

int letraexiste(char letra) {

for(int j = 0; j < strlen(palavrasecreta); j++) {

if(letra == palavrasecreta[j]) {

return 1;

}

}

return 0;

}

int chuteserrados() {

int erros = 0;

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

if(!letraexiste(chutes[i])) {

erros++;

}

}

return erros;

}

Lembre-se que quanto menor a função, mais fácil dela ser entidade e reutilizada.

## Ifs ternários

Com essa função em mãos, basta agora fazemos uma sequência de ifs, imprimindo o boneco de acordo com o número de erros. Por exemplo, os ifs para 0 e 1 erros:

int erros = chuteserrados();

if(erros == 0) {

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | \n");

printf(" | \n");

printf(" | \n");

printf(" | \n");

printf(" | \n");

printf("\_|\_\_\_ \n");

}

if(erros == 1) {

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | (\_) \n");

printf(" | \n");

printf(" | \n");

printf(" | \n");

printf(" | \n");

printf("\_|\_\_\_ \n");

}

// e assim por diante...

A ideia não é ruim. Mas nosso código ficará bem extenso. Perceba que no fim, o que queremos fazer é imprimir ou não um determinado caractere, dependendo do valor da variável erros. Vamos isolar esses caracteres no próprio printf(), fazendo uso da máscara "%c":

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | %c%c%c \n", '(', '\_', ')');

printf(" | %c%c%c \n", '\\', '|', '/');

printf(" | %c \n", '|');

printf(" | %c %c \n", '/', '\\');

printf(" | \n");

printf("\_|\_\_\_ \n");

printf("\n\n");

Agora, os caracteres do lado direito são os que precisamos imprimir de acordo com a variável erros. Veja o primeiro que aparece, o '(', por exemplo. Ele faz parte da cabeça, e portanto precisamos imprimi-lo se erros >= 1. Caso contrário, podemos imprimir "vazio". Veja que temos um if bem curto: se erros >=1, imprime ")", senão "". Esse tipo de if, que tem uma condição, um valor de retorno no caso verdadeiro, e outro no caso de falso, é possível de ser feito em uma só linha. Chamamos isso de **if ternário**.

A sintaxe de um if ternário é a seguinte: (condicao ? valor verdadeiro : valor falso). Repare no ponto-de-interrogação separando a condição do valor verdadeiro, e o dois-pontos separando o valor usado no caso que a condição é falsa. Esse tipo de if é usado "inline", ou seja, na mesma linha de outra instrução. Vamos usar junto com o printf(), por exemplo. Repare no código abaixo, quebrado em várias linhas para facilitar a legibilidade:

printf(" | %c%c%c \n",

(erros >=1 ? '(' : ' '), // Aqui o if ternario

'\_',

')'

);

A máquina fará esse if na hora de executar o printf e, dependendo da avaliação da condição, ele retornará um valor ou outro para ser impresso. Podemos fazer isso para todas as partes do nosso boneco, cada um de acordo com a sua respectiva quantidade de erros. Olhe (e digite) esse código com carinho, pois ele é extenso e cheio de caracteres:

int erros = chuteserrados();

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | %c%c%c \n", (erros>=1?'(':' '),

(erros>=1?'\_':' '), (erros>=1?')':' '));

printf(" | %c%c%c \n", (erros>=3?'\\':' '),

(erros>=2?'|':' '), (erros>=3?'/': ' '));

printf(" | %c \n", (erros>=2?'|':' '));

printf(" | %c %c \n", (erros>=4?'/':' '),

(erros>=4?'\\':' '));

printf(" | \n");

printf("\_|\_\_\_ \n");

printf("\n\n");

Pronto. Veja como o uso do if ternário nos ajudou a economizar boas linhas de código. Imagine repetirmos esse boneco um monte de vezes? Nosso arquivo ficaria gigante.

## Últimos detalhes

Vamos agora exibir uma mensagem cada vez que ele chutar uma letra, dizendo se ele acertou ou não. Vamos colocar esse código dentro da função chute(), que é a função que pega o chute do jogador. Saber se uma letra existe ou não na palavra secreta é fácil: basta invocar a função que extraímos há pouco tempo: letraexiste():

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

if(letraexiste(chute)) {

printf("Você acertou: a palavra tem a letra %c\n\n",

chute);

} else {

printf("\nVocê errou: a palavra NÃO tem a letra %c\n\n",

chute);

}

chutes[chutesdados] = chute;

chutesdados++;

}

Agora precisamos exibir a mensagem de ganhou ou perdeu para o usuário, afinal o jogo está acabando sem dizer o que aconteceu. Toda essa lógica deve existir logo após o do-while principal. Ali, basta invocarmos novamente a função ganhou() e imprimir a mensagem de sucesso que queremos. Vamos aproveitar e exibir uma caveira caso ele tenha perdido:

if(ganhou()) {

printf("\nParabéns, você ganhou!\n\n");

printf(" \_\_\_\_\_\_\_\_\_\_\_ \n");

printf(" '.\_==\_==\_=\_.' \n");

printf(" .-\\: /-. \n");

printf(" | (|:. |) | \n");

printf(" '-|:. |-' \n");

printf(" \\::. / \n");

printf(" '::. .' \n");

printf(" ) ( \n");

printf(" \_.' '.\_ \n");

printf(" '-------' \n\n");

} else {

printf("\nPuxa, você foi enforcado!\n");

printf("A palavra era \*\*%s\*\*\n\n", palavrasecreta);

printf(" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n");

printf(" / \\ \n");

printf(" / \\ \n");

printf("// \\/\\ \n");

printf("\\| XXXX XXXX | / \n");

printf(" | XXXX XXXX |/ \n");

printf(" | XXX XXX | \n");

printf(" | | \n");

printf(" \\\_\_ XXX \_\_/ \n");

printf(" |\\ XXX /| \n");

printf(" | | | | \n");

printf(" | I I I I I I I | \n");

printf(" | I I I I I I | \n");

printf(" \\\_ \_/ \n");

printf(" \\\_ \_/ \n");

printf(" \\\_\_\_\_\_\_\_/ \n");

}

Pronto. Nosso jogo está terminado. E veja o quanto você precisou aprender para fazê-lo.

É hora da versão final do jogo, igual no vídeo.

O código final ficará assim:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include "forca.h"

char palavrasecreta[TAMANHO\_PALAVRA];

char chutes[26];

int chutesdados = 0;

int letraexiste(char letra) {

for(int j = 0; j < strlen(palavrasecreta); j++) {

if(letra == palavrasecreta[j]) {

return 1;

}

}

return 0;

}

int chuteserrados() {

int erros = 0;

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

if(!letraexiste(chutes[i])) {

erros++;

}

}

return erros;

}

int enforcou() {

return chuteserrados() >= 5;

}

int ganhou() {

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(!jachutou(palavrasecreta[i])) {

return 0;

}

}

return 1;

}

void abertura() {

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n");

printf("/ Jogo de Forca \*/\n");

printf("/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\n\n");

}

void chuta() {

char chute;

printf("Qual letra? ");

scanf(" %c", &chute);

if(letraexiste(chute)) {

printf("Você acertou: a palavra tem a letra %c\n\n", chute);

} else {

printf("\nVocê errou: a palavra NÃO tem a letra %c\n\n", chute);

}

chutes[chutesdados] = chute;

chutesdados++;

}

int jachutou(char letra) {

int achou = 0;

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

if(chutes[j] == letra) {

achou = 1;

break;

}

}

return achou;

}

void desenhaforca() {

int erros = chuteserrados();

printf(" \_\_\_\_\_\_\_ \n");

printf(" |/ | \n");

printf(" | %c%c%c \n", (erros>=1?'(':' '), (erros>=1?'\_':' '), (erros>=1?')':' '));

printf(" | %c%c%c \n", (erros>=3?'\\':' '), (erros>=2?'|':' '), (erros>=3?'/': ' '));

printf(" | %c \n", (erros>=2?'|':' '));

printf(" | %c %c \n", (erros>=4?'/':' '), (erros>=4?'\\':' '));

printf(" | \n");

printf("\_|\_\_\_ \n");

printf("\n\n");

for(int i = 0; i < strlen(palavrasecreta); i++) {

if(jachutou(palavrasecreta[i])) {

printf("%c ", palavrasecreta[i]);

} else {

printf("\_ ");

}

}

printf("\n");

}

void escolhepalavra() {

FILE\* f;

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

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtddepalavras;

fscanf(f, "%d", &qtddepalavras);

srand(time(0));

int randomico = rand() % qtddepalavras;

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

fscanf(f, "%s", palavrasecreta);

}

fclose(f);

}

void adicionapalavra() {

char quer;

printf("Você deseja adicionar uma nova palavra no jogo (S/N)?");

scanf(" %c", &quer);

if(quer == 'S') {

char novapalavra[TAMANHO\_PALAVRA];

printf("Digite a nova palavra, em letras maiúsculas: ");

scanf("%s", novapalavra);

FILE\* f;

f = fopen("palavras.txt", "r+");

if(f == 0) {

printf("Banco de dados de palavras não disponível\n\n");

exit(1);

}

int qtd;

fscanf(f, "%d", &qtd);

qtd++;

fseek(f, 0, SEEK\_SET);

fprintf(f, "%d", qtd);

fseek(f, 0, SEEK\_END);

fprintf(f, "\n%s", novapalavra);

fclose(f);

}

}

int main() {

abertura();

escolhepalavra();

do {

desenhaforca();

chuta();

} while (!ganhou() && !enforcou());

if(ganhou()) {

printf("\nParabéns, você ganhou!\n\n");

printf(" \_\_\_\_\_\_\_\_\_\_\_ \n");

printf(" '.\_==\_==\_=\_.' \n");

printf(" .-\\: /-. \n");

printf(" | (|:. |) | \n");

printf(" '-|:. |-' \n");

printf(" \\::. / \n");

printf(" '::. .' \n");

printf(" ) ( \n");

printf(" \_.' '.\_ \n");

printf(" '-------' \n\n");

} else {

printf("\nPuxa, você foi enforcado!\n");

printf("A palavra era \*\*%s\*\*\n\n", palavrasecreta);

printf(" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n");

printf(" / \\ \n");

printf(" / \\ \n");

printf("// \\/\\ \n");

printf("\\| XXXX XXXX | / \n");

printf(" | XXXX XXXX |/ \n");

printf(" | XXX XXX | \n");

printf(" | | \n");

printf(" \\\_\_ XXX \_\_/ \n");

printf(" |\\ XXX /| \n");

printf(" | | | | \n");

printf(" | I I I I I I I | \n");

printf(" | I I I I I I | \n");

printf(" \\\_ \_/ \n");

printf(" \\\_ \_/ \n");

printf(" \\\_\_\_\_\_\_\_/ \n");

}

adicionapalavra();

}