Sobre o autor

Sou Yak's Vinicios Silva Souza, um estudande e entusiasta de tecnologia, boa leitura.
Email: vinicios.sousa909@gmail.com

Phone: (63) 99235 - 8700

Github: RoboCopGay

instagram: @_yaks.h

Sobre o livro

Antes de começar, este livro é destinado a iniciantes no mundo da programação, por isso todos os termos apresenta aqui estão sendo explicados da forma mais simples possível, mas os conhecimentos aqui apresentados irão ate qualquer iniciante em linguagem C (mesmo que você já saiba programar em outra linguagem).

Por favor, se você encontrar qualquer erro ortográfico ou em relação aos conhecimentos apresentados, envie-me email avisando, ou faça um pull-request no repositório do livro no github.

Repositório do livro no github: http://github.com/RoboCopGay/c_para_seres_humanos.book

Noções básicas

O que é “C”?

C é uma linguagem de programação… *“mas o que é linguagem de programação?”* …eu sei que é quase impos você ter chegado até este livro sem saber o que é linguagem de programação, mas, caso você não saiba, é “a form falar com o computador”. Você escreve o que quer que ele faça em um arquivo e ele vai fazer, e o C é só uma form se fazer isso. Existe uma infinidade de linguagens por aí, mas eu estou aqui para lhes mostrar essa que é conside por muitos uma das melhores linguagens de todos os tempos, e eu estou sendo inserido nesse “muitos”.

Como o C funciona?

O C é uma linguagem compilada… *“mas o que é isso?”*

> Basicamente, significa que o c traduz o que você escreve em um arquivo para uma liguagem que só o computad entende.

…E é considerado uma linguagem de *médio nível* (alguns o consideram uma linguagem de *baixo nível*), e com isso estou referindo-me à qualidade do C, mas ao nível de proximidade com o hardware (a parte física do computa Quanto mais próximo do hardware, mais baixo é o nível e essa característica do C o torna a linguagem mais indic para fazer aplicações de sistema (programas que manipulam o hardware) e aplicações gráficas (jogos, editores imagem…).

Só para se ter uma ideia do poder do C, vou listar alguns *softwares* feitos nessa linguagem:

> todos os softwares listados são **open source**, dessa forma vocês podem ter certeza de que foi mesmo feito em além de poderem editar o código, se quiserem…

- Blender - Modelador 3D e engine de jogos.
- Linux - Núcleo das distribuicões linux.
- Gimp - Editor de imagens.
- Darwim - Núcleo do Mac OSX
- Vlc - Reprodutor de Vídeos

Existe uma infinidade de aplicativos feitos em C, mas como o foco aqui é ensinar C (e não citar aplicativos feitos em eu vou prosseguir…

Qual é a história do C?

Resumidamente, o C foi criado na década de 1970 por Ken Thompson e Dennis Ritchie para reprogramar o UNIX, era escrito em assembly.

O C é uma evolução da linguagem B que foi influenciada pela linguagem BCPL. No início, a linguagem C despadronizada, isso significa que cada compilador de C usava uma *"versão"* diferente, então, em 1983 a A resolveu padronizar o C para que ele funcionasse mais coerentemente em compiladores diferentes, e não foi só a A que padronizou, a ISO também já fez isso.

Como se instala o tal "compilador" C?

GNU C Collection

O gcc, já vem com uma gama de ferramentas já inclusas, como o compilador C (`gcc`) e o compilador C++ (`g++`).

Linux

Se você usa uma distribuição Linux ou BSD, provavelmente já está instalado, mas, caso não esteja (o que eu du muito), é só usar o gerenciador de pacotes para instalar.

Debian

```
sudo apt install gcc
```

Red Hat

```
sudo dnf install gcc
    sudo yum install gcc
```

Arch Linux

```
sudo pacman -S gcc
```

Mac OSX

Se você não tem homebrew, rode:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

> Instalando gcc:

```
brew install gcc
```

Windows

Se você não tem o chocolatey, rode no **PowerShell em modo administrador**:

```
Set-ExecutionPolicy Bypass -Scope Process -Force
    iex((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

> Instalando gcc

```
choco install mingw -y
```

Tiny C Compiler

Uma outra opção de compilador muito interessante é o tcc é um compilador independente com o intúito de g
códigos pequenos, mas não é recomendado para aplicações grandes, mas para estudo é uma ótima opção.

Linux

Debian

```
sudo apt install tcc
```

Red Hat

```
sudo dnf install tcc
    sudo yum install tcc
```

Arch Linux

```
sudo pacman -S tcc
```

Mac OSX

```
brew install tcc
```

Windows

```
choco install tcc
```

Introdução à sintaxe do C

Hello mundo em C

> Hello_mundo.c

```
#include <stdio.h>            // importação de biblioteca externa

    int main () {                 // declaração da função main
       printf("Hello mundo!!\n"); // função que escreve coisas na tela
       return 0;                  // retorno da função
    }                             // fim da função
```

Compilando o arquivo

> linux/BSD/Mac OSX

```
gcc Hello_mundo.c -o hello
```

> Windows

```
gcc Hello_mundo.c -o hello.exe
```

Todos os exemplos e desafios são compilados da mesma forma: `gcc <arquivo.c> -o <nome do binário>`

> Caso o arquivo não compile com o `gcc` use o `g++`

Executando o arquivo (lembre-se de estar no mesmo diretório do arquivo compilado)

> linux/BSD/Mac OSX

```
./hello
```

> Windows

```
hello.exe
```

> Saída

```
Hello mundo!!
```

> Todos os exemplos e desafios são executados da mesma forma mostrada acima, apenas troque o `hello` pe
> nome do binário gerado.

No programa apresentado, nós vimos a estrutura básica de todo programa em C, no qual temos a importaçã
módulo `stdio.h` (conjunto de funções para saída e entrada de dados), a função `main` (função que é chamad
execução do programa), e os comandos da função que serão executados (tudo o que está entre `{` e `}`), que no n
caso é o `printf` (função de saída de dados que escreve texto na tela), e por fim o `return` que diz para o sist
operacional se ocorreu tudo como o planejado na execução da função `main`.

> Notem que todos os comandos dentro do bloco (tudo o que está entre `{` e `}`) da função `main` estão separad
> por `;`, caso você se esqueça desse caractere, o seu programa não será compilado.

A função `main` é essencial para o funcionamento de programas escritos em C, pois ela é a função que é chamad
execução do programa.

Mas, caso você ainda não tenha entendido, eu vou explicar de uma maneira mais simples.

> Imagine que Terezinha, uma cozinheira muito habilidosa, vai preparar um ovo frito. A primeira coisa que ela faz pegar uma frigideira com óleo e colocar no fogo. Depois, ela vai quebrar o ovo dentro da frigideira e jogar s Depois de todos os passos feitos ela dá uma olhada para saber se o ovo queimou, ou se tudo ocorreu cor desejado.

Terezinha simboliza o sistema operacional executando um programa feito em C: a frigideira é a função `main`, os ovo óleo, o sal e a fritura do ovo são todos os comandos executados pela função. Terezinha sabe se o ovo está queim ou não porque a frigideira dá um retorno (quando o ovo está mais escuro, ela sabe que queimou): esse é o pape comando `return` da função `main`.

> A partir daqui, você vai se deparar com diversos códigos soltos (para economizar espaço), mas quando for test los na sua máquina, coloque-os dentro da função `main`:

```
#include <stdio.h>

    int main(){
       <comandos apresentados>
       return 0;
    }
```

O `;` é o que delimita o fim de um comando no C e ele ignora os espaços ou quebras de linha em excesso antes do logo, isso:

```
           printf
    (
    "Hello mundo!!\n"
             )
    ;
```

É o mesmo que isso:

```
printf("Hello mundo!!\n");
```

> Então as regras para o uso do `;` são:
>
> Devem estar no final das linhas com comandos.
> Não devem estar em linhas que começarem com `#`.
> Não devem estar em linhas que terminam com chaves se essas chaves pertencem a blocos de código (po existem outras estruturas que usam chaves).

Mas mesmo sabendo disso tome cuidado com os espaços, pois em alguns casos muito específicos a falta deles p confundir o compilador, por exemplo:

```
int i = -2 - -3;
```

> não se preocupe com o `int i =`, saiba apenas que `i` é uma variável, isso será explicado mais tarde…

O `-3` é um número negativo, mas o `-` entre `-2` e `-3` é o sinal de subtração, se não houvesse espaço entre o `-` e os outros números, o programa acima não seria compilado pois o C não saberia o que você quer que ele faça.

Comentários

Comentários são “anotações ou esclarecimentos” escritas(os) no código para descrever a função de algo e geralm são úteis quando se quer analisar algum código antigo seu, ou o código de outra pessoa. Os comentários sempre ignorados pelo compilador, eles são apenas para auxiliar o programador.

```
// comentários de uma única linha
    /*
       comentários
       de
       multiplas
       linhas
    */
```

É sempre aconselhável o uso de comentários, principalmente se você pretende fazer um projeto *open source* (com dito, projetos de código aberto).

Variáveis

> Imagine um armário com diversos espaços para guardar coisas, alguns só podem guardar esferas, outros podem pra guardar bonecas, outros só podem guardar cubos e cada espaço desses possui um nome.
>
> Toda vez que alguém quer um espaço no armário deve pedir para o dono, mas o dono só reserva o espaço pedir da maneira correta, que é:
>
> `<tipo do espaço> <nome do espaço> com <coisa que queremos no espaço> dentro`
>
> Exemplos:
>
> `esfera bola_de_futbol com` :soccer: `dentro`
>
> `boneca gemeas com` :dolls: `dentro`
>
> `cubo dado com` :game_die: `dentro`

O armário é a memória do seu computador, os espaços são as variáveis e o nome em cada espaço é o nom variável, que só consegue armazenar tipos específicos de dados, no exemplo são `esfera`, `cubo` ou `boneca`, situação descrita para a reserva de um espaço é a declaração:

```
// <tipo do espaço>  <nome do espaço>  com  <coisa que queremos no espaço> dentro
       int               numero           =    80                                ;
```

No exemplo acima, reservei um espaço que só guarda números inteiros (`int`) com o nome `numero` e com o valor dentro. E no C, existem 3 tipos primitivos, ou seja, 3 tipos de dados básicos, o `int`, o `float` e o `char`.

```
// <tipo do espaço>  <nome do espaço>  com  <coisa que queremos no espaço> dentro
       int              numero           =    80                                ;
       char             caractere        =    'A'                               ;
       float            numero_real      =    99.9                              ;
```

Variáveis do tipo `int` recebem números sem ponto, como `2` , `8` ou `234` enquanto as do tipo `float` recebem núm com ponto como `2.5` , `8.3` ou `23.0` , já variáveis do tipo `char` recebem um caractere, **apenas um** , logo, se te colocar um `"hello"` ou até mesmo um `"h"` , ele vai retornar um erro, pois todo e qualquer caractere entre `"` é string enquanto um `char` é um único caractere entre `'` , não se preocupe com as strings, por enquanto…

E se você deseja alterar o valor da variável, só precisa colocar o nome da variável recebendo o valor:

```
int i = 5; // declaração da variável "i" valendo "5"

    i = 92;    // agora a variável vale 92
```

Esse exemplo acima serve para todos os tipos primitivos, mas lembre-se de colocar valores do tipo certo na varia Se você declarou uma variável inteira, na hora de alterar o valor, tem que trocar por um inteiro.

Além disso, também existe o tipo `double` que é descendente do tipo `float` , mas com mais capacidade de espa *“Mas como assim espaço? Números não são infinitos?”* … Os números são infinitos sim, mas a memória ran computador não é, e mesmo que fosse, seria um desperdício liberar um espaço infinito para uma única variável, e todas as variáveis dentro do C tem um espaço limitado.

Cada variável ocupa uma certa quantidade de bytes na memória ram:

```
/*
    um byte tem 8 bites e um bite só pode ser 0 ou 1, isso quer dizer que
    sempre que o computador reserva 1 byte ele está reservando 8 espacinhos
    com zeros e ums.
    */

    int inteiro = 0; /* -> 4 bytes -> você só consegue colocar
                          números de -2147483648 a 2147483647,
                          pois se o número estiver fora desse
                          intervalo ele teria mais de 4 bytes.
                    */

    char caractere = 'A'; /* 1 byte -> só aceita um caractere,
                             porque um caractere ocupa um
                             byte.
                          */

    float real = 3.14; /* 4 bytes -> só suporta valores entre
                          10E-38 e 10E38 (isso significa 10
                          vezes 10 elevado a -38 a 38, o "E"
                          substitui o "vezes 10 elevado a"
                          para simplificar para o computador).
                       */

    double real_2 = 10E49; /* 6 bytes -> o double tem mais
                                  espaço que um float
```

```
                                    e por isso pode suportar
                                    números entre 10E-4932 e
                                    10E4932
                                */
```

> Essas quantidades demonstradas acima não são iguais em todas as arquiteturas (tipo de processador), isto qu
> dizer que se o seu computador é de 32 bits o tamanho das variáveis pode ser diferente de um de 64 bits, log
> para que você tenha certeza do tamanho delas (em bytes) é só usar o `sizeof` :

```
int inteiro;
    int tmh_inteiro = sizeof inteiro; // tamanho da variável inteiro
```

> Caso você não queira criar uma variável unicamente para pegar o seu tamanho é possível usar o `sizeof` pa
> pegar o tamanho do tipo diretamente:

```
int tmh_inteiro = sizeof (int); // tamanho da variável inteiro
```

> Note que o tipo está entre parênteses, isso é obrigatório ou o C vai achar que você está se referindo a un
> variável.

E a galera que já conhece um pouco de programação deve estar se perguntando *"Mas e os boleanos? No C não e
verdadeiro e falso?"* sim, mas no C o `int` faz esse papel, sendo que o **0** equivale a **falso** e o **1** equiva
verdadeiro.

As variáveis em C (e acho que em todas as linguagens) têm algumas regras quanto à escolha do seu nome, e
regras são:

- Variáveis não podem iniciar com numeros;
- Variáveis não podem ter espaços (substitua os espaços por `_`);
- Variáveis só podem conter letras, números e travessões (evite usar letras com acento também);
- Variáveis não podem ser iguais à palavras reservadas.

> Palavras reservadas do C:

```
auto break case char if const continue  default do double else enum
    extern float for goto if int long register return short signed void
    sizeof static struct  switch  typedef union unsigned volatile while
```

Sendo assim, variáveis com nomes como `2letras` , `char` , `jo%ao` ou `peso da pedra` estão erradas, mas variáveis c
`_2letras` , `Char` , `joao` ou `peso_da_pedra` estão certas, e tome muito cuidado com o uso de maiúsculas e minúscu
pois o C as diferencia, portanto, `char` é uma palavra reservada, mas `Char` não é.

Conflito entre tipos

Um problema (na minha opinião) do C é a forte tipagem, que significa que os tipos têm que ser respeitados a to
custo, logo, se queremos que um dado seja transformado em outro, precisamos fazer conversões de tipos.

```
int Um = (int) 1.5; // apenas o 1 será atribuído
```

Quando você atribui a uma variável o valor que pertence a outro tipo o C vai converter isso para o tipo da variável,
isso não irá acontecer quando você fizer operações matemáticas, por isso é sempre bom converter para o tipo certo

```
int i = 1.5e-9;   // apenas o 1 será atribuído
    float f =  i / 3; /*
                        o resultado dessa divisão seria 0.033...
                        Mas como os dois  números são inteiros o
                        resutado  é  0, e por  isso é  atribuído
                        o valor 0.0 à variável f
                      */
```

> Portanto, sempre use variáveis do mesmo tipo para operações matemáticas, caso sejam de tipos diferentes use
> conversão de tipos.

Modificadores de tipo

E mais uma vez falaremos de tipos primitivos, como havíamos visto, os tipos primitivos tem tamanhos diferente
memória, e estes tamanhos podem ser expandidos ou reduzido.

Long

O `long` alonga (expande) a capacidade de variáveis do tipo `int` , `float` e `double` .

> Lembrando que os valores de tamanho variam de computador para computador.

```
int inteiro = 0;              // 4 bytes
    long int l_inteiro = 0;       // 8 bytes

    double real = 10E49;          // 6 bytes
    long double l_real = 10e49;   // 8 bytes
```

E para alcançar o máximo de tamanho de uma variável para a sua arquitetura use o `long long int` ou simplesm
`long long` .

Short

O `short` encurta a capacidade de variáveis do tipo `int` .

```
int inteiro = 0;              // 4 bytes
    short int s_inteiro = 0;      // 2 bytes
```

Signed e unsigned

`signed` e `unsigned` significam respectivamente “com sinal” e “sem sinal”.

```
int c = 90;
    int i = +90;
    int j = -90;
```

Sempre que você declara um número, ele por padrão é `signed`, portanto suporta números negativos e positivos, `unsigned` só suporta números positivos.

```
int inteiro = 0;              // intervalo: -2147483648 a 2147483647
    unsiged int us_inteiro = 0;    // intervalo: 0           a 4294967295
```

Entrada e saída de dados

Você já viu anteriormente uma forma de saída de dados: o `printf`:

```
printf("Hello mundo!!\n");
```

> Sem este `\n`, caso você escreva outra coisa os dois irão aparecer juntos.

Caractere de scape (“\”)

O caractere de scape, no C, é o `\` e ele dá “poderes” ao seu texto, pode ser usado em variáveis do tipo `char` e strings, mas para ilustrá-lo eu irei representá-lo sempre dentro de um `printf`.

\n

```
printf("\n1ª linha\n2ª linha\n3ª linha\n");   // \n: quebra de linha.
```

Esse `\n` é uma quebra de linha, ou seja, sempre que tiver um `\n` o `printf` vai pular uma linha e escrever o estiver na frente.

> saída:
>
> ```
> 1ª linha
>     2ª linha
>     3ª linha
> ```

\t

```
printf("\tjoao"); // \t: espaçamento ou tabulação (efeito da tecla "tab").
```

Esse `\t` é uma tabulação, o que estiver a frente dele irá se deslocar para a direita (->).

> saída:
>
> ```
>     joao
> ```

\b

```
prinft("joao\b");  // \b: apagua um caractere da linha (efeito da tecla "backspace").
```

Esse `\b` é um backspace, o caractere anterior a ele será apagado.

> saída:
>
> ```
>  joa
> ```

\r

```
printf("coisas mais coisas\r outras coisas"); // \r: elimina tudo o que está antes dele na linha.
```

Esse `\r` vem de “remove”, todos os caracteres da mesma linha e anteriores a ele serão apagados.

> saída:
>
> ```
>  outras coisas
> ```

\v

```
printf("coisas\voutrascoisas\vjoao\v."); // \v: tabulação vertical.
```

O `\v` vai quebrar a linha assim como o `\n` , mas em vez de iniciar a nova no inicio da linha, ele inicia no “final
anterior, formando uma “escadinha”.

> saída:

```
  coisas
           outrascoisas
                       joao
                           .
```

\”

```
printf("\"joao\" é um nome feio); // \": exibe as aspas duplas.
```

Exibe as aspas duplas (`"`), pois se você escrever simplesmente `"` o C vai achar que aquele é o fim da string.

saída:

```
"joao" é um nome feio
```

\’

```
printf("it\'s estranho"); // \': exibe as aspas simples ou apótrofos.
```

Exibe a aspa simples (que alguns chamam de apóstrofo).

saída:

```
it's estranho
```

****

```
printf("isso é uma contra-barra: \\");     // \\: exibe a contra-barra
```

saída:

```
isso é uma contra-barra: \
```

printf

“Mas e se eu quiser imprimir uma variável?” … é só usar a formatação de texto do `printf` … *“Mas como se usa is* … para imprimir uma variável `int` é só escrever um `%i` ou `%d` dentro da string. Se for um `char` , escreva `%c` string, se for `float` , escreva `%f` , se for uma notação científica (geralmente usada no tipo `double`), escreva `%E` estiver usando o “e” maiúsculo) e `%e` para o “e” minúsculo, após escrever a formatação desejada, é só lista

variáveis separando por vírgula ligo após a string… *“Eu não entendi nada do que tu disse!”* …Relaxa… Olhe o exer
e suas dúvidas em relação a isso irão desaparecer:

```
int numero = 90;
    char caractere = 'A';
    float real = 9.23;
    double real_em_dobro_E = 9.4E13;
    double real_em_dobro_e = 9.4e13;

    printf("numero inteiro: %i", numero);
    printf("numero real: %f", real);
    printf("numero real notação cientifica com \"E\": %E", real_em_dobro_E);
    printf("numero real notação cientifica com \"e\": %e", real_em_dobro_e);
    printf("caractere: %c", caractere);
```

> saída:

```
numero inteiro: 90
    numero real: 9.23
    numero real notação cientifica com "E": 9.4E13
    numero real notação cientifica com "e": 9.4e13
    caractere: A
```

scanf

O scanf é semelhante ao printf, mas serve para ler dados:

```
int numero;

    scanf("%i", &numero);
```

Tá, eu sei que você está se perguntando *“e esse* `&` *serve pra que?”* Esse `&` diz para o `scanf` colocar o valor no l
da memória onde está o número. O `&` simboliza um endereçamento de memória, o `scanf` coloca o valor direto no
da memória onde está a variável.

E como você pode perceber o `%i` se refere a um número inteiro. Todos os tipos de variáveis são simbolizados p
símbolos (`%i` , `%c` , `%f` …) do `printf` .

Também é possível ler várias variáveis com um único comando:

```
int numero;
    char caractere;
    float real;

    printf("digite um numero um caractere e um numero real: \n");
    scanf("%i %c %f", &numero, &caractere, &real);

    printf("\nnumero inteiro: %i", numero);
    printf("numero real: %f", real);
```

```
    printf("caractere: %c", caractere);
```

> Na hora de ler um `char`, às vezes o `scanf` buga, isso ocorre quando ele recebe lixo do teclado, você só preci
> ler a variável duas vezes, isso geralmente ocorre com `char`, mas se acontecer com outro tipo, a resolução para
> problema é a mesma:

```
fflush(stdin); // esse comando vai limpar o lixo da memória
```

> Saída:
>
> ```
> digite um numero, um caractere e um numero real:
>     3458
>     J
>     5.8769
>
>     numero inteiro: 3458
>     numero real: 5.8769
>     caractere: J
> ```

Esses não são os únicos métodos de entrada e saída de dados, mas veremos outros em outros capítulos, esses s
bastante para prosseguirmos nossos estudos.

Formatos do printf e scanf

%s

Strings ou texto, exemplo:

```
printf("string: '%s'\n", "string");
```

%d e %i

Inteiros, exemplo:

```
printf("int: %i\n", 90);
```

%f

Reais, exemplo:

```
printf("float: %f\n", 9.3);
```

E como são números com `.` você pode formatar a saída deles, o `9.3` vai ser exibido como `9.300000`, mas eu q
que sáia `9.3`

```
printf("float: %.1f\n", 9.3);
```

> Notem que entre o `%` e o `f` existe um `.1`, isso quer dizer que só é para exibir `1` numero após a "vírgula"(que
> C é um `.`).

O protótipo é mais ou menos assim:

```
printf("%.<decimais>f\n", <numero>);
```

%c

Caracteres, exemplo:

```
printf("char: %c\n", 'A');
```

%o

Numeros octais, exemplo:

```
printf("int: %o\n", 018);
```

> Numeros octais iniciam com `0`, logo `012` é o mesmo que `10`

%u

Numeros sem sinal, exemplo:

```
printf("int: %u\n", 18);
```

%x

Numeros hexadecimais, exemplo:

```
printf("int: %x\n", 0xDB7B5);
```

> Todo hexadecimal começa com `0x`.

%l

Numeros longos (sempre acompanhado pelo tipo alongado), exemplo:

```
// Reais longos
    printf("double: %lf\n", (double) 9.3);

    // Inteiros longos
    printf("long int: %li\n", 698);
```

Putchar e puts

Basicamente o “put” significa coloque, logo, `putchar` é coloque um `char` :

```
char c = '\n';

    putchar ('j');
    putchar ('o');
    putchar ('ã');
    putchar ('o');
    putchar ( c );
    putchar ('!');
    putchar ( c );
```

> Saída:

```
joao
    !
```

E seguindo a mesma lógica, `puts` é coloque uma string (o `s` é uma abreviação).

```
char * str = "string coisada";

    puts ( "joao" );
    puts ( "!" );

    puts ( str );
```

> Saída:

```
joao
    !
    string coisada
```

> Uma particularidade do `puts` é que ele adiciona um `\n` no fim da string.

Getchar e gets

Assim como o `scanf` , o `getchar` e o `gets` , são funções para leitura de dados, mas que só servem para ler variáveis tipo `char` e strings.

É assim que se usa o `getchar` :

```
char j;

    j = getchar();
```

E o `gets` é usado assim:

```
char str [20];
    gets(str);
```

> Mesmo o `gets` sendo uma função contraindicada pela comunidade, ela ainda funciona, então caso o gcc apon erros pelo uso do `gets` , saiba que ela vai funcionar normalmente.

Fprintf e fgets

O `f` antes do `printf` significa formatação, logo, um `fprintf` é um `printf` formatado, e essa formataçã basicamente um parâmetro a mais indicando o lugar onde você quer escrever a informação.

```
fprintf(stdout, "Hello mundo!!\n"); // printf ( "Hello mundo!!" )
```

> O `stdout` é um "stream" (local para onde vai a string do `fprintf`), e o `printf` é um `fprintf` com o `stdou` como "stream", e o `stdout` é a saída padrão (a tela).

Mas também é possível enviar a saída para outros streams, dentre eles nós temos o `stderr` , que é a saída padrã erros:

```
char coisa [30];

    puts("escreva de 1 a 10 caracteres: ");
    scanf ("%s", &coisa );

    if (coisa [0] == '\0' ) {
       fprintf ( stderr, "ERROR: você não digitou nenhum caractere!");
       return 1;
    }

    if (coisa [10] != '\0' ) {
       fprintf ( stderr, "ERROR: você digitou caracteres demais!");
       return 1;
    }
```

> No programa acima são pedidos caracteres de 1 a 10, então se o caractere da posição `0` corresponder ao fim uma string (`'\0'`) quer dizer que 0 caracteres foram lidos, e se o 11º caractere (posição `10`) for o fim da stri

> (`'\0'`) quer dizer que existem mais de 10 caracteres na string.

Note que quando ocorreu um erro o valor retornado foi o `1` , indicando para o sistema operacional que aconteceu erro.

O `fgets` seria um `gets` formatado, e ele funciona da seguinte maneira:

```
char str[10];

    fgets ( stdin, 10, str ); // gets (str)
```

> O `stdin` é a entrada de dados padrão (o teclado).

“Ué? Então por que eu deveria usar esse `fgets` aí se o `gets` é mais simple?” …Muito simples, lembra que o `gets` um problema, tanto que ele é contra-indicado pelo próprio compilador? Pois é, o `fgets` não tem esse problema, po nele além de você indicar a string a ser lida e o “stream”, ele exige que você coloque o tamanho da string, a evitando colocar dados no lugar errado.

Operadores

Aritméticos

Os operadores aritméticos são os operadores matemáticos e são expressos da seguinte maneira em C:

```
n + N // Adição            -> soma os dois números;
    n - N // Subtração         -> subtrai os dois números;
    n * N // Multiplicação     -> multiplica dois números;
    n / N // Divisão           -> divide dois números;
    n % N // resto da divisão  -> retorna o resto da divivão entre dois números.
```

> para evitar erros sempre faça operações com números de tipos iguais.

E nunca se esqueça que em expressões numéricas existe uma ordem de precedencia, logo, `6+4/2` é `8` e não isso acontece porque assim como na matemática é resolvida primeiro a multiplicação (`4/2`) e depois é somado esse valor.

Odem de precedencia:

- parênteses (`(` `)`)
- multiplicação (`*`), divivão (`/`) e resto (`%`)
- adição (`+`) e subtração (`-`)

```
6+4/2    //  ->  8
    (6+4)/2  //  ->  5
```

Atribuição

O operadores de atribuição são formas simplificadas de atribuir valores… *“Não entendi…“*

Isso é uma atribuição:

```
numero = 89;
```

E caso eu queira que este número valia ele mesmo + 1 eu faço:

```
numero = numero + 1;
```

Mas para poupar esforços o C também aceita:

```
numero += 1;
```

E isso vale para qualquer operação:

```
numero += 2; // numero = numero + 2
    numero -= 3; // numero = numero - 3
    numero *= 7; // numero = numero * 7
    numero /= 2; // numero = numero / 2
```

Além desses também existe os operadores de incremento e decremento:

```
numero ++; // numero = numero + 1
    numero --; // numero = numero - 1
```

> os vistos acima são denominados de pós incremento, pois a variável só recebe o valor depois de retorna-lo *que?”* … Observe:

```
int numero = 89;
    printf("%i\n", numero++ );
```

> Saída:

```
89
```

“Pera! mas ele não deveria ser 90?” … A variável `numero` só é incrementada depois de retornar o valor dela, isso dizer que ela só é incrementada depois dessa parte do programa, mas se você usar o pré-incremento:

```
int numero = 89;
    printf("%i\n", ++numero );
```

> Saída:

```
90
```

```
numero ++; // pós-incremento
    numero --; // pós-decremento

    ++ numero; // pré-incremento
    -- numero; // pré-decremento
```

Relacionais

Os operadores lógicos são todos aqueles que testam uma expressão relacional e dizem se ela é verdadeira ou falsa

> lembrando que no C verdadeiro é 1 e falso é 0.

```
n == N // igual            -> testa se n é igual a N;
    n != N // diferente        -> testa se n é diferente de N;

    n <  N // menor que        -> testa se n é menor que N;
    n >  N // maior que        -> testa se n é maior que N;

    n <= N // menor ou igual   -> testa se n é menor ou igual a N;
    n >= N // maior ou igual   -> testa se n é maior ou igual a N;
```

Exemplo simples:

```
printf("%i\n", 1 < 2);
    printf("%i\n", 1 > 2);
    printf("%i\n", 1 != 2);
    printf("%i\n", 1 == 2);
```

> Saída:

```
1
    0
    1
    0
```

> E lembrem-se que os tipos dos dados sendo testados tem que ser o mesmo.

Eles serão usados por vocês nas estruturas condicionais, laços de repetição e com o **operador ternário**.

Lógicos

Os operadores lógicos são usados para assimilar operações que retornam valores lógicos… *“Como assim?”* … usados para operações verdadeiras e falsas.

> João pede ao seu pai um fone de ouvido e um celular, mas o seu pai só dá o fone de ouvido, e ele fica insatisfe pois ele queria as duas coisas.

João só ficaria satisfeito (verdadeiro) se ele ganhasse o celular e o fone de ouvido, como ele só ganhou o fone ficou insatisfeito (falso).

O operador usado no exemplo acima é o “and” (`&&`) que só é “verdadeiro” se as duas opções forem verdade exemplo:

```
printf("%i && %i = %i\n", 1 < 5, 2 > 0, 1 < 5   &&   2 > 0);
    printf("%i && %i = %i\n", 1+80 < 5, 2 > 0, 1+80 < 5   &&   2 > 0);
```

> Saída:

```
1 && 1 = 1
    0 && 1 = 0
```

Os operadores lógicos são o `&&` , que equivale a “and”(“e”), já visto anteriormente, o `||` , que equivale a “or”(“ou”) `!` , que equivale a “not”(“não”).

> considere os uns e zeros abaixo apenas o resultado de alguma operação relacional…

	&&		resultado final
1	&&	1	1
0	&&	1	0
1	&&	0	0
0	&&	0	0

	||		resultado final
1	||	1	1
0	||	1	1
1	||	0	1
0	||	0	0

!		resultado final
!	1	0
!	0	1

Exemplo:

```
printf("%i && %i = %i\n", 1 < 5, 2 > 0, 1 < 5   &&   2 > 0);

    printf("%i || %i = %i\n", 1 < 5, 2 > 6, 1 < 5   ||   2 > 6);

    printf("!(%i && %i) = %i\n", 1 < 5, 2 > 6, !(1 < 5   &&   2 > 6));
```

> Saída:

```
1 && 1 = 1
    1 || 0 = 1
    !(1 && 0) = 1
```

Ternário ou condicional

```
int  numero = 80;
    char imparOuPar = (  numero % 2 == 0  ) ? 'p'               : 'i'             ;
    //                ( <expreção logica> ) ? <se for Verdade>  : <se for Mentira>
```

O código acima representa um uso simples do operador ternário e eu sei que você deve estar um pouco confuso isso, mas eu explico:

> Antes de mais nada, saiba que **todo número par tem o resto da divisão por 2 igual 0** , então, no exemp acima, caso `( numero % 2 == 0 )` seja verdadeiro, o operador vai retornar `'p'` de par, casoa expressão seja fals ele vai retornar `'i'` de impar, portanto, o C vai testar a expressão lógica entre `(` e `)` se essa expressão verdadeira, o valor da operação vai ser o que está entre `?` , e aso contrário, o valor será o que está após `:`

Outro exemplo do uso seria:

```
int nota = 6;
    char status = ( nota >= 7 ) ? 'p' : 'r';

    printf("João %s de ano", (status=='p')? "passou" : "reprovou" );
```

No código acima, se a nota do João for maior ou igual a 7, o programa escreve `"João passou de ano"` na tela, se ele escreve `"João reprovou de ano"` .

Chegou a hora de praticar!

Agora chegou a hora de praticar, e não pule essa parte, pois o seu aprendizado só é absoluto se você praticar, e
para potencializar o seu aprendizado em C e em qualquer linguagem de programação:

> Dicas:
>
> > Se tiver dificuldade em algo na hora da resolução do exercício, primeiro volte ao assunto antes de consulta
> > resposta;
> > Sempre escreva todo o código: não use o `ctrl+C` e `ctrl+V` enquanto ainda está aprendendo, pois quan
> > você escreve, está acostumando o seu cérebro com a sintaxe da linguagem;
> > Sempre que você conseguir resolver o desafio, antes de pular para o próximo, tente resolvê-lo de novo
> > outra maneira;
> > Caso não consiga resolver, veja a resposta e depois tente fazer de novo de outra maneira;
> > Crie seus próprios desafios para dificultar os que estão aqui.

Todos os desafios serão resolvidos e explicados linha a linha, exceto o último de cada rodada, pois esse você va
que resolver sozinho, obrigatoriamente, para tentar provar para si mesmo que aprendeu e se você não conseguir,
de novo os conteúdos anteriores e tente novamente. Caso você passe para a próxima parte sem resolvê-lo,
dificuldades posteriores em outros assuntos.

Desafio 1

Faça uma calculadora onde o usuário digite dois números (reais) e no final ele exiba todas as operações matemá
com esses números:

> saída:

```
digite um número: 3
    digite outro número: 4

    3 + 4 = 5
    3 - 4 = 5
    3 * 4 = 12
    3 / 4 = 0.75

    A divisão inteira entre 3 e 4 é 0 e o resto dessa divisão é 3
```

Resposta

Antes de mais nada, nós temos que digitar nossa estrutura padrão:

```
#include <stdio.h>
    int main (){

    return 0;
    }
```

Depois, nós temos que pedir dois números para o usuário:

```
printf("digite um número: ");
    printf("digite outro número: ");
```

Agora, iremos ler os dois números, mas antes, temos que criar as variáveis que vão guardar esses números:

```
float numero, outro_numero; // dessa forma criamos várias variáveis do mesmo tipo de uma vez
```

Agora, nós podemos ler os números:

```
printf("digite um número: ");
    scanf("%f", &numero);

    printf("digite outro número: ");
    scanf("%f", &numero);
```

Finalmente, iremos exibir os resultados:

```
printf("\n");
    printf("%f + %f = %f\n", numero, outro_numero, numero + outro_numero);
    printf("%f - %f = %f\n", numero, outro_numero, numero - outro_numero);
    printf("%f * %f = %f\n", numero, outro_numero, numero * outro_numero);
    printf("%f / %f = %f\n", numero, outro_numero, (float) numero / (float) outro_numero);
    printf("\n");
    printf("A divisão inteira entre %f e %f é %i e o resto dessa divisão é %i\n",
       numero, outro_numero, numero / outro_numero, numero % outro_numero);
```

E o código final ficou assim:

```
#include <stdio.h>
    int main (){

       // criando variáveis que serão usadas
       int numero, outro_numero; // dessa forma criamos várias variáveis do mesmo tipo de uma vez

       // lendo variáveis
       printf("digite um número: ");
       scanf("%i", &numero);

       printf("digite outro número: ");
       scanf("%i", &outro_numero);

       // exibindo variáveis
       printf("\n");
       printf("%i + %i = %i\n", numero, outro_numero, numero + outro_numero);
       printf("%i - %i = %i\n", numero, outro_numero, numero - outro_numero);
       printf("%i * %i = %i\n", numero, outro_numero, numero * outro_numero);
       printf("%i / %i = %.2f\n", numero, outro_numero, (float) numero / (float) outro_numero);
       printf("\n");
```

```
        printf("A divisão inteira entre %i e %i é %i e o resto dessa divisão é %i\n",
        numero, outro_numero, numero / outro_numero, numero % outro_numero);

    return 0;
    }
```

Desafio 2

Faça uma calculadora na qual o programa peça dois números e depois uma operação (a escolha deve ser entre so
subtração).

> Saída:

```
Digite um número inteiro: 8
    Digite outro número inteiro: 2
    Digite a operação [+/-]: +

    A soma entre 8 e 2 é 10
```

Resposta

A primeira coisa que devemos fazer é obviamente escrever a estrutura padrão:

```
#include <stdio.h>

    int main (){

        return 0;
    }
```

Agora, temos que declarar as variáveis que irão guardar os dados:

```
int numero, outro_numero;
    char operacao;
```

E temos que pedir os dados para o usuário:

```
printf("Digite um número inteiro: ");
    scanf("%i", &numero);

    printf("Digite outro número inteiro: ");
    scanf("%i", &outro_numero);

    printf("Digite a operação [+/-]: ");
    fflush(stdin);
```

```
    scanf("%c", &operacao);
```

> no meu caso ocorreu aquele bug do `scanf` que eu mencionei no capítulo de entrada e saída de dados, mas ca
> no seu não aconteça apague a linha com o `fflush` .

Agora, iremos testar se a operação escolhida foi soma ou subtração e depois salvar o resultado em outra vari
(`res`):

```
int res = ( operacao == '+' ) ? numero + outro_numero : numero - outro_numero ;
    //   ...se operacao for +            some          senao   subtraia
```

E finalmente exibimos os resultados:

```
printf ( "a %s entre %i e %i é %i\n",
       ( operacao == '+' ) ? "soma" : "subtração",
       numero,
       outro_numero,
       res
    );
```

O código final ficou assim:

```
#include <stdio.h>

    int main (){

       // declarando variáveis
       int numero, outro_numero;
       char operacao = '+';

       // lendo variáveis
       printf("Digite um número inteiro: ");
       scanf("%i", &numero);

       printf("Digite outro número inteiro: ");
       scanf("%i", &outro_numero);

       printf("Digite a operação [+/-]: ");
       // resolvendo bug do scanf
       scanf("%c", &operacao);
       // lendo variável
       scanf("%c", &operacao);

       // calculando resultado
       int res = ( operacao == '+' ) ? numero + outro_numero : numero - outro_numero ;
       //   ...se operacao for +            some          senao   subtraia

       // exibindo resultado
       printf ( "a %s entre %i e %i é %i\n",
             ( operacao == '+' ) ? "soma" : "subtração",
             numero,
             outro_numero,
```

```
            res
        );

        return 0;
    }
```

Desafio 3

Faça um programa que leia 3 números e diga quantos deles são ímpares ou pares e quantos deles são divisíveis p
(se divididos por 3 o resto tem que ser 0).

> Saída:

```
Digite 3 números: 2 1 3
    2 são ímpares, 1 é par e 1 é divisível por 3
```

Você já deve ter percebido que este é complicado, mas calma… É só pensar bem, e uma dica, explore bem o terr
antes.

Resposta

A primeira coisa que iremos fazer é declarar e ler os números (depois de escrever a estrutura padrão é claro) :

```
int n1, n2, n3; // Essa é a forma de declarar várias variáveis ao mesmo tempo

    printf ("Digite 3 números: ");
    scanf ("%i %i %i", &n1, &n2, &n3);
```

Agora nós iremos declarar contadores para os ímpares, pares e divisíveis por 3 e iniciá-los com `0` (se não fizer
suas variáveis vão receber lixos da memória):

```
int impar = 0, par = 0, divPor3 = 0; // essa é a forma de inicializar várias variáveis ao mesmo tempo
```

Declarados os contadores, iremos testar os números pares e ímpares:

```
// se o número for divisível por 2 incremente par senão incremente impar
    ( n1 % 2 == 0 )? par ++: impar ++;
    ( n2 % 2 == 0 )? par ++: impar ++;
    ( n3 % 2 == 0 )? par ++: impar ++;
```

Agora testamos os divisíveis por 3:

```
divPor3 = ( n1 % 3 == 0 )? divPor3 + 1: divPor3;
    divPor3 = ( n2 % 3 == 0 )? divPor3 + 1: divPor3;
```

```
    divPor3 = ( n3 % 3 == 0 )? divPor3 + 1: divPor3;
```

E por fim exibimos os valores:

Aqui nós temos duas opções, exibimos os dados de forma preguiçosa:

```
printf ( "%i são ímpares\n", impar);
    printf ( "%i são pares\n", par);
    printf ( "%i são divíveis por 3\n", divPor3);
```

> E nesse caso, quando o contador vale 0 ou 1, vai ficar “0 são " ou "1 são ".

Ou tentamos adaptar a resposta para que ela respeite os plurais e singulares e assim criando um programa
inteligente:

> Ambos os códigos de exibição dos dados funcionarão da mesma forma, basicamente temos que exibir u
> resposta assim:
>
> `<contador> <são (se plural)|| é (se singular)> <info que o contador se refere>`
>
> E caso o contador for `0` essa resposta tem que ser assim: `nenhum é <info que o contador se refere>`
>
> mas para economizar linhas de código essas duas formas de frase teriam que ser uma só, logo, eu temos q
> montar um esqueleto assim:
>
> `<contador><caractere auxiliar><string indicando a quantidade> <info a que o contador se refere>`
>
> Então a string do `printf` ficou assim: `"%i%c%s %s\n"`
>
> Agora que temos o “esqueleto” da resposta, temos que dar valores a esses campos e o primeiro é o contado
> `par` , `impar` e `divPor3`)
>
> O segundo é o caractere auxiliar e ele tem que apagar o contador caso ele for nulo (igual a `0`) com o caracte
> `\b` , caso contrário ele insere um espaço (‘ ‘):
>
> `( <contador> == 0 ) ? '\b' : ' '`
>
> O terceiro é a string indicando a quantidade, e nesse caso temos três opções, se o contador for 0 essa string te
> que ser `"nenhum é"` , se o contador não for `0` ele testa se o contador é igual a 1, e caso for verdadeiro essa stri
> vai valer `"é"` (indicando que é singular), senão ele obviamente é plural (contador maior que `1`) então essa stri
> vai valer `"são"` :
>
> `(<contador> == 0)? "nenhum é" : ( ( <contador> == 1 )? "é" : "são" ) .`
>
> Por fim é só testar se é plural ou singular e colocar a informação a que o contador se refere no plural ou singular:
>
> `(<contador> > 1)? <info no plural> : <info no singular>`
>
> O “esqueleto” final do printf ficou assim:

```
printf ( "%i%c%s %s",
            <contador>,                                          // %i
            (<contador> == 0)? '\b' : ' ',                       // %c
            (<contador> == 0)? "nenhum é" :
                  ( ( <contador> == 1 )? "é" : "são" ),          // %s
            (<contador> > 1)? <info no plural> : <info no singular>  // %s
    );
```

E é assim que fica a exibição dos resultados seguindo o esqueleto acima:

```
// Hora de exibir os resultados

    printf(
       "%i%c%s %s,",

       impar,
       (impar == 0 )? '\b' : ' ',

       (impar == 0 )?  "nenhum é" :
          ( (impar == 1 )? "é" : "são" ) ,

       (impar > 1)? "ímpares": "ímpar"
    );

    printf(
       " %i%c%s %s",

       par,
       (par == 0 )? '\b' : ' ',
       (par == 0 )? "nenhum é" :
          ( (par == 1 )? "é" : "são" ),

       (par > 1)? "pares": "par"
    );

    printf(
       " e %i%c%s %s\n",

       divPor3,
       (divPor3 == 0 )? '\b' : ' ',

       (divPor3 == 0 )? "nenhum é" :
          ( (divPor3 == 1 )? "é" : "são" ),

       (divPor3 > 1)? "divisíveis por 3": "divisível por 3"
    );
```

E o código final ficou assim:

```
#include <stdio.h>
    #include <stdlib.h>
    int main(int argc, char **argv){

       // declaração e leitura de variáveis
       int n1, n2, n3;
```

```
    printf ("Digite 3 números: ");
    scanf ("%i %i %i", &n1, &n2, &n3);

    // declaração e inicialização dos contadores
    int impar = 0, par = 0, divPor3 = 0;

    // se o número for divisível por 2 incremente par senão incremente impar
    ( n1 % 2 == 0 )? par ++: impar ++;
    ( n2 % 2 == 0 )? par ++: impar ++;
    ( n3 % 2 == 0 )? par ++: impar ++;

    // se é divisível por 3 incremente senão não incremente
    divPor3 = ( n1 % 3 == 0 )? divPor3 + 1: divPor3;
    divPor3 = ( n2 % 3 == 0 )? divPor3 + 1: divPor3;
    divPor3 = ( n3 % 3 == 0 )? divPor3 + 1: divPor3;

    // Hora de exibir os resultados

    printf(
       "%i%c%s %s",

       impar,
       (impar == 0 )? '\b' : ' ',

       (impar == 0 )?  "nenhum é" :
          ( (impar == 1 )? "é" : "são" ) ,

       (impar > 1)? "ímpares": "ímpar"
    );

    printf(
       "%i%c%s %s",

       par,
       (par == 0 )? '\b' : ' ',
       (par == 0 )? "nenhum é" :
          ( (par == 1 )? "é" : "são" ),

       (par > 1)? "pares": "par"
    );

    printf(
       " e %i%c%s %s\n",

       divPor3,
       (divPor3 == 0 )? '\b' : ' ',

       (divPor3 == 0 )? "nenhum é" :
          ( (divPor3 == 1 )? "é" : "são" ),

       (divPor3 > 1)? "divisíveis por 3": "divisível por 3"
    );

    return 0;
}
```

Desafio 4

Faça um programa que leia 3 números e diga qual é o maior e qual é o menor, e se a pessoa digitar números iguais tem que avisar “foram digitados números iguais”.

> Saída:

```
Digite 3 números:2
    3
    3

    O número 1 é o menor
    O número 3 é o maior

    foram digitados números iguais
```

Este você tem que fazer sozinho, todas as coisas necessárias para fazê-lo foram ensinadas. Boa sorte e se conseguir fazer, releia os capítulos anteriores com muito cuidado, e depois tente de novo.

Noções avançadas

Arrays

Arrays são variáveis com vários espaços… *“Como assim?”* … Lembra do armário das variáveis? No caso do array vez de reservar um espaço, você pede vários espaços de uma vez, tipo:

> `cubo dado [] com` :game_die: , :game_die: , :game_die: `dentro`

```
int algarismos [] = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 0 };
```

E também é possível acessar um item específico do array adicionando a posição entre `[` e `]`.

```
int algaridobro desmos [] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    int tres = algarismos[2]
```

Agora, a variável `tres` está com o item `3` , *“Mas o 3 está da 3º posição!”* , eu entendo, esse é um erro que iniciante comete, a questão é que a contagem começa do `0` :

```
//                  [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
    int algarismos [] = {  1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 0};
```

E desta forma é possível modificar um valor do array:

```
algarismos[9] = 9;
```

Mas para modificar o array inteiro é necessário modificar item por item, portanto, a seguinte forma não funcionará:

```
int algarismos [] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    algarismos = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Os arrays não podem ser impressos ou atribuídos, isso significa que todos os itens tem de ser imprimidos um por ur

```
a [] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    printf (
       "%i, %i, %i, %i, %i, %i, %i, %i, %i, %i\n",
       a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[0]
    );
```

E antes que alguém pergunte *“Só existe array inteiro?”*, não, você pode fazer arrays com qualquer tipo primitivo.

Se você prestou atenção no capítulo sobre variáveis você deve conhecer o comando `sizeof`, que diz a quantidad bytes de uma variável, se você usar com arrays também:

```
int i [3];
    printf ("o tamanho de "i" é %i", sizeof i);
```

“Mas e se eu quiser saber a quantidade de elementos do meu array?” … você só precisa dividir o tamanho do a pelo tamanho do tipo desse array… *“Como assim?”* … O tamanho de uma variável `int` é o mesmo do tipo `int`, log tamanho do tipo é o tamanho de uma variável com esse tipo…

```
int i;
    int array_i [8];

    int tmh_i = sizeof i;           // tamanho de i
    int tmh_ari = sizeof array_i;   // tamanho de array_i

    int qnt_elem = tmh_ari / tmh_i; // quantidade de elementos
```

Ou, se você for preguiçoso igual eu:

```
int array_i [8];

    int qnt_elem = sizeof array_i / sizeof array_i[0]; // quantidade de elementos
```

> lembrem-se de que `array_i` é um array do tipo `int`, ou seja, tem elementos do tipo `int`, logo, se eu usar tamanho de um desses elementos eu também vou ter acesso ou tamanho do tipo…

Strings

Uma string é uma cadeia de caracteres ou um array de caracteres ou um "vetor" de caracteres.

"Mas se é um array, por que eu preciso ler essa parte se você já explicou os arrays?" … Muito simples, as strings arrays e possuem todas as características de arrays, mas elas têm algumas particularidades.

```
char Char = 'A';          // isso é um caractere
    char String [] = "coisa"; // isso é uma cadeia de caracteres

    // essa é a maneira de tradicional de escrever a string
    char other_str [] = {'c', 'o', 'i', 's', 'a', '\0'};
```

> tanto a variável `String` quanto a `other_str` são strings, mas, caso você use a segunda forma, não se esqueça `\0` no fim, pois este caractere é o que delimita o fim de uma string.

Caso não queira dar valor para a string, você deverá dizer qual o seu tamanho (assim como em qualquer outro tip array), exemplo:

```
char String [10];
```

> Caso você necessite usar a forma acima, lembre-se de colocar um numero a mais no comprimento da string, po o último caractere é reservado para simbolizar o fim de uma string, por isso esta string só tem 9 caracteres válido

"Tá, mas como eu dou um valor para ela?" … Você pode fazer isso de três maneiras.

A primeira forma é a tradicional para dar valores a um array:

```
char s [13];

    s[0] = 'c';
    s[1] = 'o';
    s[2] = 'i';
    s[3] = 's';
    s[4] = 'a';
    s[5] = 's';
    s[6] = '\0';
```

> E só pra relembrar: NÃO SE ESQUEÇA DO `\0` … Tô parecendo até flashback de naruto com esse caractere …

Na segunda, você tem que importar a biblioteca `string.h` e depois usar a função `strcpy` para atribuir o valor, d maneira:

```
#include <string.h>

    char str[13];
    strcpy(str, "coisas");
```

> O `include` acima pode ficar tanto dentro da função `main` quanto fora, só é importante colocar antes do uso
> função `strcpy` .

E a terceira, você vai ter que esperar pelo capítulo de ponteiros para entender, logo, use as mostradas acima
enquanto.

Note que você não é obrigado a colocar a quantidade exata de caracteres na string, mas não coloque uma string m
que o tamanho tamanho estipulado.

“Legal, mas se eu quiser ler uma string? Ou imprimir uma string?” … Muito simples, é só usar o `%s` no `scanf`
`printf` , mas em relação ao `scanf` temos uma leve diferença:

```
char str [20];
    scanf("%s", &str);

    printf("a string lida foi: %s\n", str);
```

Uma coisa que eu sei que você ficou na dúvida no capítulo sobre variáveis foi o por que de `'A'` ser diferente de `"A`

```
char Char = 'A';
    char String = "A";
```

‘A’ é diferente de “A”, pois a string tem sempre um caractere nulo no fim:

	[0]	[1]
Char	‘A’	
String	‘A’	‘\0’

Os blocos e o escopo

Até agora eu apenas mencionei o que é um bloco lá no inicio do livro para explicar a função `main` , mas acho que a
não foi o suficiente para encerrar o assunto.

A primeira característica de um bloco, que a propósito já foi abordada, é que ele é uma sequencia de comandos e
`{` e `}` , mas uma coisa que você não sabia é que ele pode ser usado sem a necessidade de uma estrutura:

```
int numero;

    {
       numero = 89;
```

```
    }

    numero += 67;
```

“mas que diferença isso fez para o programa?” , neste caso acima nenhuma, mas os blocos podem ser úteis pa organização do código e também para o isolamento de variáveis:

```
int n = 46;
    {
       char a = n;
       printf("char: %c\n", a);
    }
    printf("char: %c\n", a);
```

O programa a cima não irá compilar por que a variável `a` só existe dentro do bloco, logo, o segundo `printf` nã conseguir imprimir a variável.

Isso acontece por causa de uma coisa que chamamos de escopo, e o escopo determina o nível de acesso de variável no código, isso significa a variável `a` do código anterior só podia ser acessada dentro do bloco ond declarada, e é o que chamamos de variável local.

```
int coisa = 0;

    int main () {

       int outra_coisa = 90;

       // bloco
       {
          int A = 100;
       }
       return 0;
    }
```

O escopo do código anterior seria assim:

> Trate o “X” como “pode acessar” o bloco dentro da função `main` é o “bloco” da tabela e tudo o que está fora função `main` é a área “global”.

	bloco	**`main`**	**global**
`coisa`	X	X	X
`outra_coisa`	X	X	
`A`	X		

> A variável `coisa` é global porque está fora de todos os blocos possíveis, portanto pode ser acessada em qualqu bloco do programa, enquanto a variável `outra_coisa` é local da função `main` , logo, todo e qualquer bloco den do `main` pode ter acesso a ela, e a variável `A` só existe dentro do bloco isolado.

```
int a = 78973;
    {
       char a = 'a';
       printf("char a : %c\n", a);
    }
    printf("int a : %i\n", a);
```

> Saída:
>
> ```
> char a : a
>    int a : 78973
> ```
>
> O que aconteceu aqui foi que a variável `a` do tipo inteiro foi declarada fora do bloco, logo, ela existia també dentro do bloco, até que outra variável `a` foi declarada dentro do bloco também, e o valor do `a` de dentro do blo não sob-escreve o `a` de fora do bloco porque ela só existe dentro do bloco, e quando o bloco acaba a variável do tipo `char` deixa de existir e o `a` inteiro continua existindo…

Eu sei que é um pouco confuso, mas isso acontece simplesmente por que as variáveis de escopos diferentes declaradas áreas diferentes da memória.

E nunca se esqueça que essas regras vistas neste capítulo serve para todo e qualquer bloco…

Condicionais

Até agora nós fizemos códigos mais sequenciais, onde todos os comandos eram executados e a única forma decisão que usamos foi o operador ternário.

```
int n;

    printf("digite um número: ");
    scanf("%i", &n);

    // com o ternário
    printf("o número %i é %s ", n, (  n % 2 == 0   ) ?     "par"    :  "ímpar"   );
    //                           (operação lógica) ? caso verdade : caso falso

    // com condicionais
    printf("o número %i é ", n)
    if (n % 2 == 0) {         // se n for par
       printf("par \n");      //   escreva "par"
    } else {                  // senão
       printf("ímpar \n");    //   escreva "ímpar"
    }
```

Note que os condicionais são muito mais intuitivos, e por isso são mais fáceis de usar que o ternário… *"Ué? Então que você mostrou o ternário primeiro?"* …Porque eu precisava de uma estrutura que não exigisse um conhecim sobre blocos e escopo e que tornasse o você mais preparado para esse assunto, pois se você entendeu o ternário com certeza entendeu os condicionais.

“Mas com o ternário ficou muito mais curto!” …Sim, mas e se sempre que o número for ímpar ele tenha também pedir outro número ao usuário?…

> Não se questione sobre esta funcionalidade a mais, ela é só uma forma de dificultar um pouco o exemplo.

```
// com ternário
    int n;

    printf("digite um número: ");
    scanf("%i", &n);

       (  n % 2 == 0  )? printf("o número %i é par", n) : printf("digite outro número: ");
    //   se n for par         diga que ele é par      senão        peça outro número

    int reserva = n;
    n = ( n % 2 == 0 ) ?   n       :      scanf("%i", &reserva);
    //   se n for par    n é n   senão      leia outro número

    n = reserva;
```

> antes que você se pergunte o por quê de eu ter criado a variável `reserva`, se o código estivesse assim:
>
> ```
> n = ( n % 2 == 0 ) ? n : scanf("%i", &n);
> ```
>
> O `scanf` iria ler o valor, colocar dentro do n, mas quando ele finalizasse esta instrução ele retornaria um `1` ou u `0` para informar se deu certo ou não, e esse feedback do `scanf` substituiria o valor lido por ele, e sempre que número fosse ímpar o `n` seria `1` ou `0`, então a variável `reserva` serve de reserva para o valor de `n`.

```
// com condicionais
    int n;

    printf("digite um número: ");
    scanf("%i", &n);

    if (n % 2 == 0) {                    // se n for par
       printf("o número %i é par", n);   // diga que ele é par
    } else {                             // senão
       printf("digite outro número: ");  // peça outro número
       scanf("%i", &n);                  // e leia esse número
    }
```

Note que o código feito com condicionais ficou muito mais organizado e simples de entender, além de elimin necessidade da variável auxiliar `reserva` …

Existem 3 formas de fazer um condicional no C:

```
int condicao = (67 != 5);  /*
                                  e se você prestou atenção
                                  no capítulo de operadores
                                  sabe que o valor dessa
```

```
                                variável é 1.
                              */
    int outra_condicao = (8 > 2);

    // simples

    if ( condicao ) {
       // comandos
    }

    // composto
    if ( condicao ) {
       // comandos
    else {
       // outros comandos
    }

    // aninhado
    if ( condicao ) {
       // comandos
    else if ( outra_condicao ) {
       // comandos
    } else {
       // outros comandos
    }
```

> Mas o bloco não é obrigatório para o uso do condicional, caso você queira um `if` mais compacto:

```
if (condicao) /* comando */;
```

> Caso você precise executar mais de um comando você vai ter que usar um bloco, mas você usar a forma acim
> em conjunto com as anteriores, logo, o seguinte código é válido:

```
int i = 3847;

    if (i % 2 == 0) printf("O número é par!\n");

    else if ( ( (float) i / 3.0 == 0.0 ) && ( i % 2 == 1 ) ) {
       printf("O número é ímpar e divisível por 3!");
       printf("\n");
    }

    else printf("O número é ímpar!");
```

Estruturas de repetição

Estruturas de repetição são estruturas que permitem que você repita comandos, e isso te permite automatizar algu
coisas como atribuir valor a arrays ou fazer contagem…

While

```
// exibindo uma contagem até 5

    // sem estruturas de repetição
    printf("%i\n", 1);
    printf("%i\n", 2);
    printf("%i\n", 3);
    printf("%i\n", 4);
    printf("%i\n", 5);

    // com uma estrutura de repetição
    int contador = 1;
    while (contador <= 5){          // enquanto o contador for menor ou igual 5
       printf("%i\n", contador);    //   escreva o contador
       contador ++;                 //   e incremente o contador
    }
```

> No caso acima temos duas formas de exibir uma contagem… *"Mas deu a mesma quantidade de linhas, então n vale a pena aprender esse negócio difícil aí, a primeira opção é mais simples!!"* … E se ao invés de contar até nós precisássemos contar até mil? Você ainda acha aceitável escrever isso sem usar uma estrutura de repetição

A estrutura usada é o `while`, que significa "enquanto", basicamente ele funciona assim:

```
condicao = (2 != 3);
    while ( condicao ) {   // enquanto a condição for verdade execute
       // comandos
    }
```

E se a condição for verdadeira o `while` vai executar os comandos do bloco, mas se ela for falsa ele vai sair do (estrutura de repetição).

E assim como nos condicionais, se você precisar

No caso da contagem foi necessário adicionar `1` ao contador, pois se a `condicao` sempre for verdadeira o progr vai entrar em um loop infinito, e foi isso que aconteceu com o nosso exemplo anterior, pois o `2` sempre será difer do `3`.

Mas isso não quer dizer que loops infinitos sejam sempre ruins,Digamos que agente queira que um programa dados, mas não sabemos a quantidade exata de vezes ler, então criamos um loop infinito e damos um "flag"(uma fo de interrupção do loop) a ele, no nosso exemplo flag será a resposta para a pergunta "Deseja continuar?", ca pessoa digite "n" loop será interrompido.

Eu vou ilustrar a situação acima de duas maneiras usando o `while`.

```
// usando o teste lógico do while
    {
       char flag = 's';

       while (flag == 's'){
          printf("Deseja continuar? [s/N] ");
          scanf("%c", &flag);
       }
```

```
    }

    // usando o break
    {
       char flag;

       while (1) {
          printf("Deseja continuar? [s/N] ");
          scanf("%c", &flag);

          if (flag != 's') break; // se a resposta for não interrompa
       }
    }
```

> O break é um comando que interrompe loops

E assim como nos condicionais o `while` também tem uma forma compacta:

```
int cont = 0;
    while (cont <= 10) printf("%i\n", cont++);
```

Do..while

A estrutura popularmente conhecida como `do .. while` é basicamente um `while` de cabeça para baixo.

```
// exibindo uma contagem até 5
    int i = 1;

    // while

    while ( i <= 5 )         // enquanto ( i menor ou igual a 5 )
       printf ( "%i\n", i++ );  //     escreva i

    // do..while
    do {                        // faça {
       printf ( "%i\n", i );     //    escreva i
    } while ( i<=5 );            // } enquanto ( i menor ou igual a 5 )
```

O `do .. while` funciona da mesma forma que o `while` , com uma única diferença, ele faz o teste lógico no final, logo executa o que está no bloco e só depois testa a condição:

```
int i = 90;

    do {
       printf("%i\n", i);
    } while ( i <= 10 );
```

> Saída:

```
90
```

“Ué? Por que ele imprimiu?” , Porque ele faz o teste lógico (`i <= 10`) no fim da execução do bloco, se este fosse `while` comum o teste seria feito antes, e só executaria o bloco se este teste fosse verdadeiro.

“Aah! Então é inútil usar este laço!!” … Sinto discordar, mas este laço foi criado para agilizar algumas tarefas, lembr código com flag? Que vimos no capítulo anterior?

```
char flag = 's';

    while (flag == 's'){
       printf("Deseja continuar? [s/N] ");
       scanf("%c", &flag);
    }
```

Para que o código acima funcione nós somos obrigados o valor `'s'` à variável `flag` , com o `do .. while` isso n necessário:

```
char flag;

    do {
       printf("Deseja continuar? [s/N] ");
       scanf("%c", &flag);
    } while (flag == 's');
```

E o código com `while` e sem usar o teste lógico (o que usamos o `break` para sair do loop) é um `do .. while` es manualmente!

For

O `for` é uma forma mais automatizada de loop, ele é mais usado para contagem, mas também é possível usá-lo flag.

Em uma comparação direta com o `while` :

```
// exibindo uma contagem até 5

    // while
    {
       int i = 0;
       while (i <= 5) printf("%i\n", i++);
    }

    // for
    {
       for ( int i = 0; i <= 5; i++) printf("%i\n", i);
    }
```

“Caramba!! O que aconteceu aqui?” … Se você não estiver entendido o código acima eu irei mostrar da fo tradicional:

```
// exibindo uma contagem até 5

    // while
    {
       int i = 0;            // dando o valor 0 a i
       while (i <= 5) {      // enquanto i menor ou igual a 5
          printf("%i\n", i); // escreva i
          i ++;              // incremente i
       }
    }

    // for
    {
       for ( int i   =   0; i <= 5;     i++      ) {
    // para      i entre 0    e  5  incremente i
          printf("%i\n", i);
       }
    }
```

“Ainda não entendi o `for` “ … Note que temos três espaços entre os parenteses separados por `;` , no primeiro esp você declara uma variável, no segundo você digita o teste lógico, e no terceiro você digita um incremento.

Basicamente o `for` é uma gambiarra do `while` , ainda utilizando o exemplo anterior observe um for escrito no estil um `while` :

```
int i = 0;

    for ( ; i <= 5 ; ){
       printf("%i\n", i);
       i++;
    }
```

> Os espaços entre `;` podem ficar em branco, mas convenhamos que é um tanto inútil usar um `for` des maneira…

E as formas diversas de usar o `for` são essas:

```
// usar variável já existente
    int i;
    for (i = 1; i <= 5; i++) printf("%i\n", i);

    // criando variável local exclusiva para o uso do for
    for ( int i = 1; i <= 5; i++) printf("%i\n", i);

    // forma while
    int i = 1;
    for ( ; i <= 5; ) printf("%i\n", i++);

    // forma de loop infinito com flag
    int i = 1;
```

```
    for (;;) {
       printf ("%i\n", i++);
       if ( i == 5 ) break;
    }
```

Deu pra perceber que o `for` é bem eclético não é mesmo? Mas definitivamente a forma em que ele é mais útil tradicional:

```
for ( int <nome da variável> = <valor inicial>; <teste lógico>; <incremento>) {
       // comandos
    }

    // forma compacta
    for ( int <nome da variável> = <valor inicial>; <teste lógico>; <incremento>) /* comando */;
```

Ponteiros

Os ponteiros ou pointers no inglês, são variáveis que guardam endereços de memória.

Lembra do `&` antes da variável no `scanf` ?

```
int n;
    scanf("%i", &n);
```

Este `&` indica um endereço de memória da variável `n` , e para guardar este endereço em uma variável, é neces que agente crie um ponteiro:

```
int n = 9;
    int * ponteiro_n = &n;
```

Este `*` antes do nome da variável diz para o C que esta variável vai guardar endereços de memória… *"Mas po colocar um tipo se a variável vai guardar só endereços? Por acaso endereço tem tipo?* … Não é bem assim, temos dar um tipo ao ponteiro porque ele também tem tamanho, e para que consiga armazenar o endereço de uma vari ele tem que ter o mesmo tamanho.

"Tá, mas ainda não entendi a utilidade desse negócio!" , acho que essa mentalidade vai mudar assim que descobrir que um array é um ponteiro que aloca vários espaços na memória.

Existem algumas regras sobre o uso de ponteiros, por exemplo:

```
int i = 90;

    int * p = &i; // o ponteiro "p" agora aponta para a variável "i"

    printf("%i\n", *p);
```

> Quando damos o endereço de memória de uma variável para um ponteiro, nós dizemos que esse ponteiro apon para essa variável

No exemplo acima, declaramos um ponteiro `p` apontando para `i`, agora nós podemos ter acesso ao valor de apenas adicionando um `*` antes do `p`, e com isso conseguimos exibir o valor de `i` na tela com o `printf`.

E usando esse `*` também podemos alterar o valor de `i`:

```
int i = 90;

    int * p = &i;
    *p = 89;
```

> Só que nós estamos alterando o valor diretamente na memória, e uma prova disso é que se você incrementar ponteiro `p`, terá acesso a outro endereço de memória.

```
int i = 90;
    int * p = &i;

    p ++;

    *p = 89;
```

Quando você executar o código acima irá ocorrer um erro de segmentação (quando o programa tenta acessar memória que não pertence a ele) ou se não ocorrer erro, quer dizer que ele acessou um espaço de mem desconhecido, e quando este ultimo ocorre o valor que está nesse espaço é um lixo do sistema ou o local onde alocada outra variável.

```
int a[] = { 2, 4, 5, 6};

    printf("%i\n", a[1]); // 4

    a ++;
    printf("%i\n", * a);  // 4
```

Como um array é um ponteiro, nós podemos usar o array como um ponteiro, *"Mas por que você incrementou o `a` a de exibi-lo?"*, porque se eu usá-se o endereço original, o valor exibido seria o `2`, pois o endereço de memória ser se refere ao primeiro valor.

Mas o método que usei anteriormente não é muito adequado já que uma vez que você incremente o array ele es apontando para outra posição e assim você tem que decrementar toda vez, o que não é prático, então, a forma indicada para isso seria:

```
int a[] = { 2, 4, 5, 6};

    printf("%i\n", a[1]);     // 4
```

```
    printf("%i\n", *(a+1) );  // 4
```

> Desta forma o valor de `a` não será alterado.

Isso também serve para atribuir valor aos itens de um array.

```
int a[] = { 2, 4, 5, 6};

    a[1] = 90;

    *(a+2) = 56
```

Notem que o índice (o valor entre `[` e `]`), é somado a `a` , isso acontece porque um array cria uma fila de espaço mesmo tipo, uma do lado da outra, por isso `*(a+3)` é o mesmo que `a[3]` .

Alocação dinâmica (arrays dinâmicos)

Em alguns casos, precisamos de mais espaço do que a variável comum para guardar dados, e para esses ca geralmente usamos arrays, mas e se durante a execução eu necessite de um array maior… *"É só criar um array m e usar ele para a manipulação do novos dados!"* … Isso pode até funcionar, mas não é recomendável, pois seria desperdício de memória.

Para resolver isso nós podemos alocar a quantidade de memória que queremos (em bytes) e usar um ponteiro este endereço de memória, e se quisermos um espaço maior, é só realocar a memória deste ponteiro, assim poderemos aumentar e diminuir o tamanho do array.

> E como prometi no capítulo sobre strings… Esta é a terceira forma de atribuir uma string:

```
char * str;
    str = "string";
```

> Isto só funciona com strings, arrays de outros tipos tem que ser atribuídos item a item.

O próprio exemplo da atribuição de uma string é um exemplo de alocação dinâmica, mas ela é feita automaticament

```
char * str;          // aqui temos um ponteiro vazio.
    str = "coisa";       /*
                            aqui nós alocamos 6 bytes na memória para
                            guardar { 'c', 'o', 'i', 's', 'a', '\0' }
                         */

    printf("%s\n", str);

    str = "outra coisa"; /*
                            aqui nós realocamos o espaço de 6 bytes
                            para 12 bytes e assim podemos guardar
                            {
                               'o', 'u', 't', 'r', 'a', ' ',
                               'c', 'o', 'i', 's', 'a'
```

```
                            }
                        */

    printf("%s\n", str);
```

Se fossemos fazer o código acima usando puramente ponteiros, nós faríamos assim:

> Lembrando que ao alocar espaços e referenciando com ponteiros, nós estamos criando arrays.

Antes de mais nada você tem que incluir o `stdlib.h` no seu arquivo (para evitar erros, sempre faça qualquer `inc` no inicio do arquivo)

```
#include <stdlib.h>          // biblioteca necessária para usar as funções de alocação.
```

> Para garantir inclua esta biblioteca em todos os exemplos a partir daqui.

Agora sim, podemos continuar…

```
    char * str;                  // aqui temos um ponteiro vazio.

    str = malloc (6);            // aqui nós alocamos 6 bytes na memória.

    // guardando dados...
    *( str + 0 ) = 'c';   // str[0] = 'c';
    *( str + 1 ) = 'o';   // str[1] = 'o';
    *( str + 2 ) = 'i';   // str[2] = 'i';
    *( str + 3 ) = 's';   // str[3] = 's';
    *( str + 4 ) = 'a';   // str[4] = 'a';
    *( str + 5 ) = '\0';  // str[5] = '\0';

    printf( str );
    putchar('\n');

    str = realloc (str, 12);        // aqui nós realocamos o espaço de 6 bytes para 12 bytes

    // guardando dados...
    str = "outra coisa";
    str[12] = '\0';

    printf("%s\n", str);

    free( str );           /*
                               essa linha vai no fim do programa e serve
                               para liberar a memória que nó alocamos,
                               para não ocorrerem erros sempre temos
                               que liberar a memória.
                           */
```

> Note que o ultimo printf está antes do `free` , pois se ele estiver depois, vai dar erro já que o espaço aloca anteriormente seria apagado.

A saída de ambos os códigos é a mesma:

```
coisa
    outra coisa
```

“Ah então eu vou sempre usar a primeira forma, porque é mais fácil!” , use, mas não se esqueça que a primeira fo só funciona com strings, para outros tipos de arrays você terá que usar a segunda forma.

Só para fixar melhor veja como funcionariam o array dinâmico com o tipo `int` .

```
    // alocando a memória que o array terá
    int * array_dinamico = malloc ( sizeof (int) * 4); /*
                                                          aqui nós alocamos um espaço que caiba 4 inteiros,
                                                          pois o nosso array inicial terá 4 posições.
                                                        */

    array_dinamico [0] = 2;
    *( array_dinamico + 1) = 3;
    array_dinamico [2] = 23;
    array_dinamico [3] = 894;

    // realocando memória para que caibam 5 posições
    array_dinamico = realloc ( array_dinamico , sizeof (int) * 5);

    array_dinamico [0] = 2;
    *( array_dinamico + 1) = 3;
    array_dinamico [2] = 23;
    array_dinamico [3] = 894;
    *( array_dinamico + 4) = 34;
```

> Lembre-se de alocar a quantidade certa de memória para o ponteiro, ao contrário dos arrays aqui você tem q saber a quantidade exata de bytes reservar, um macete muito útil é:

```
<tipo> * <variável> = malloc ( sizeof (<tipo>) * <quantidade de posições>);
```

> Desta forma a quantidade de bytes necessária será sempre respeitada.

Funções

A estrutura de uma função já foi explicada anteriormente de uma forma bastante resumida:

```
int main(){
       printf("Hello mundo!!");
    return 0;
    }
```

Basicamente uma função é uma rotina, que pode ser usada durante a execução de um código, um exemplo é a fu `printf` , que nada mais é do que um conjunto de códigos que escrevem coisas na tela, uma rotina que é execu sempre que é chamada.

A estrutura de uma função é simples:

```
//  tipo do retorno  nome     parametros
        int              soma ( int n1, int n2) {
          return n1 + n2;
       }

       int main () {
          printf("%i\n", soma( 34, 54));
          return 0;
       }
```

> Claro que você pode declarar a quantidade de parâmetros que quiser.

E se você não quiser retornar nenhum valor declare a função como `void` .

Mas se você for criar uma função, certifique-se de que ela foi declarada antes da função `main` , e se mesmo assim ainda queira que a sua função fique depois do `main` , você tem que antes declarar a função:

```
void oi () ;

    int main () {
       oi();
       return 0;
    }

    void oi () {
       printf("oi!!\n");
    }
```

E um extra sobre a função `main` é que é possível receber dados como parâmetros… *“Como assim?”* … Digamos queremos que o nosso programa escreva coisas na tela…

> Execução do nosso programa:

```
./escreva Hello mundo!!
```

> Saída:

```
Hello mundo!!
```

Os parâmetros `Hello` e `mundo!!` foram passados para a função `main` pela linha de comando, e para que agente p receber e usar esses parâmetros você precisa declará-los a área de parâmetros do `main` .

```
int main (int arg_counter, char * arg_variable []) {

       for (int i = 1; i <= arg_counter; i ++)
          printf("%s ", arg_variable[i]);

       printf("\b\n");
    return 0;
    }
```

> O `arg_counter` é a quantidade de argumentos recebidos, o `arg_variable` é um parâmetros com os argumentos, esses parâmetros ou argumentos, são strings.

> No exemplo acima usamos `arg_counter` e `arg_variable` para o nome dos parâmetros, e você pode usar o q você quiser, mas a maioria das pessoas usam `argc` (`arg_counter`) e `argv` (`arg_variable`).

Por fim, é sempre interessante saber como seria um ponteiro para uma função:

```
#include <stdio.h>

    int (* Soma) (int n1, int n2);

    int somador (int n, int n2){
      return n + n2;
    }

    int main(){
      Soma = somador;
      printf("%i + %i = %i", 2, 3, Soma(2, 3));
    return 0;
    }
```

Estruturas

Até agora você só viu estruturas padrões do próprio C, e como usá-las, mas agora você vai aprender a criar as s próprias…

Structs e unions

`struct` é o tipo de dado que cria uma estrutura própria, e é muito útil para criar “objetos” ou seja, criar variáveis várias características…

> E a galera que já conhece um pouco mais deve estar se perguntando “Mas o C é orientado a objetos?” e desde não, o máximo que você pode fazer no C é criar um tipo com espaços para armazenar dados, mas não é possív criar objetos ou classes.

```
    struct pessoa {
       char * nome;
       int idade;
```

```
        char sexo;
        float peso;
        float altura;
    }
```

Como podem ver no exemplo acima, nós criamos uma estrutura `pessoa` que pode receber um `nome`, uma `idade` `peso`, um `sexo` e uma `altura`, assim melhorando e muito nosso armazenamento de dados, *"Mas como eu p acessá-los?"*, muito simples:

```
struct pessoa joao;        // aqui nós criamos uma pessoa "joao".
    joao.nome = "Joao";        // aqui nós atribuímos "Joao" ao nome da pessoa.
```

E como você pôde notar agora existe um tipo `struct pessoa`, *"Mas, eu quero criar um tipo `pessoa`, é possível?"*, sim e para isso você vai usar o `typedef`, e ele serve para apelidar um tipo.

```
typedef int MyInt;

    MyInt inteiro;
```

Mas como nós queremos usá-lo com a nossa `struct`, temos 3 formas de usar:

Criando o `struct` antes:

```
struct p { char * name };
    typedef struct p pessoa;
```

Criando ao mesmo tempo:

```
typedef struct _p { char * name } pessoa;
```

Criando ao mesmo tempo com uma `struct` anônima:

```
typedef struct { char * name } pessoa;
```

Não importa a forma que você escolha, todas vão funcionar:

```
pessoa joao;
    joao.name = "Joao";
```

> E para evitar erros de escopo, sempre declare structs fora do `main`.

Outra estrutura muito interessante é a `union`, ela é semelhante a `struct`, mas só vai assumir uma variável qua declarada… *"Como assim?"* …observe:

```
// struct
    {
       struct p {
          char * nome;
          int idade;
       };

       // Uso

       struct p joao;
       joao.nome = "Joao";
       joao.idade = 12;
    }

    // union
    {
       union p {
          char * nome;
          int idade;
       };

       // Uso

       union p joao;
       joao.nome = "Joao";       // aqui você escolheu usar a variável nome
       joao.idade = "jumento";   // aqui você está atribuíndo "jumento" à variável nome
    }
```

> *"Ué? Como assim?"*... O que acontece é que você tem que escolher apenas uma das variáveis da `union` e outras variáveis vão apontar para a escolhida. Isso significa que se você atribuísse valor a `joao.idade` primei teria que lidar com valores inteiros na `idade.nome`.

Enum

O `enum` vem enumeração e nesse você deseje designar valores constantes para as suas estruturas.

```
typedef enum {
       true = 1,
       false = 0,
    } bool;

    bool falso = false;
```

> Só pode colocar inteiros em enums.

E acima acabamos de criar o tipo booleano no C.

Como só é possível colocar inteiros em enums, e por isso existe um macete legal para atribuir esses números:

```
typedef enum {

        zero = 0,        // zero é 0
        um,              // um é zero + 1
        dois,            // dois é um + 1
        tres,            // tres é dois + 1

        sete = 7,        // sete é 7
        oito,            // oito é sete + 1
        nove,            // nove é oito + 1

        quatro = 4,      // quatro é 4
        cinco,           // cinco é quatro + 1
        seis             // seis é seis + 1

    } por_extenso;
```

Comandos do pré-processador

O pré-processador é a ferramenta que prepara o código para a compilação, por isso existem ele tem seus próp comandos, e caso você queira curiar a versão preprocessada de um arquivo em C use o comando `gcc -E <arquiv <saida.c>`

Basicamente todos os comandos do pré-processador começam com `#` , e um desses comandos é o próprio `#incl` que usamos para importar nossas bibliotecas, que além de importar bibliotecas padrões, você pode importar s próprios arquivos.

```
#include "minhas_funcoes.c"
    #include "/home/robocopgay/biblioteca.c"
```

> Quando você usa as aspas duplas (") você pode passar o caminho para a sua biblioteca (caso ela esteja mesmo diretório coloque apenas o nome do arquivo)

Outra diretris interessante é o `#declare` , que serve para criar constantes e macros:

```
#define PI 3.14
    #define soma (n, n2) n + n2

    #define add_item ( array, item )\
    array = realloc( sizeof (array) + sizeof (item) );
```

> Note que na ultima linha da ultima macro foi usado o `\` para indicar que a próxima linha pertence a ela.

A vantagem aqui é que se precisa de uma constante, é mais útil usar o `#define` que criar uma variável, pois a vari tem que ocupar um espaço na memória, enquanto o pre-processador apenas substitue o lugar onde a macr constante é chamada pelo seu conteúdo.

E a diretris `#undef` *"desdefine"* uma macro ou constante

```
#define max 10

    int i = max*3;

    #undef max
    define max 30
```

E existe o `#if` , `#else` , `#elif` e o `#endif` , usados para condicional

```
#define MIN_SIZE 2

    #if defined(MAX_SIZE) // if -> se
      #define tamanho MAX_SIZE

    #elif MIN_SIZE > 2 // else if -> senão se
      #define tamanho 2

    #else // else -> senão
      #define tamanho MIN_SIZE+1

    #endif // end -> fim

    #include <stdio.h>
    #include <stdlib.h>

    int main (int argc, char ** argv){
      printf("%i", tamanho);
    return 0;
    }
```

Outra coisa interessantíssima é criar strings a partir de código, *"Ué? não entendi…"*, muito simples… Quando usam operador `#` dentro de uma macro ele transforma o comando em string

```
#define to_str(texto) #texto

    puts( to_str(Hello mundo!!) );
```

> Saída:

```
Hello mundo!!
```

Ou se quiser juntar duas informações use o `##`

```
#define to_str(texto) #texto
    #define str_function_template(func) str##func

    int l = str_function_template(len)("abacate");
    printf("\"%s\" tem %i letras.\n", "abacate", l);
```

> Saída:

```
"abacate" tem 7 letras.
```

Chegou a hora de praticar de novo!

Desafio 5

Faça uma calculadora onde o usuário digite dois números (reais) e no final ele pergunte qual operação matema
fazer (+, -, / ou *) e no fim ele pergunte se a pessoa deseja calcular de novo.

> Saída:

```
Digite 2 números: 2 3
    Você quer somar (+), subtrair (-), multiplicar (*) ou dividir (/)?
     +
    2 + 3 = 5

    Deseja calcular de novo? [S/n] n
```

Resposta

Primeiramente iremos declarar as variáveis necessárias:

```
int n1, n2;    // números que iremos ler
    char operacao; // operação
    int res;       // resposta
```

E iremos ler os dados necessários:

```
printf("Digite 2 números: ");
    scanf("%i %i", &n1, &n2);

    printf("Você quer somar (+), subtrair (-), multiplicar (*) ou dividir (/)? ");
    operacao = getchar();
```

Agora nós vamos efetuar os devidos cálculos:

```
if (operacao == '+')
       res = n1 + n2;

    else if (operacao == '-')
       res = n1 - n2;
```

```
    else if (operação == '/')
       res = n1 / n2;

    else res = n1 * n2;
```

E exibimos o resultado:

```
  printf("\n%i %c %i = %i\n", n1, operacao, n2, res);
```

E se você é atento notou que faltou perguntar se a pessoa que calcular de novo, mas antes de fazer esta pergunt temos que colocar o código que queremos repetir dentro de uma estrutura de repetição, mas não coloque a part declaração de variáveis:

```
  do {

        printf("Digite 2 números: ");
        scanf("%i %i", &n1, &n2);

        printf("Você quer somar (+), subtrair (-), multiplicar (*) ou dividir (/)? ");
        operacao = getchar();

        if (operacao == '+')
           res = n1 + n2;
        else if (operacao == '-')
           res = n1 - n2;
        else if (operação == '/')
           res = n1 / n2;
        else res = n1 * n2;

        printf("\n%i %c %i = %i\n", n1, operacao, n2, res);

        printf("Deseja calcular de novo? [S/n] ");

        if ( getchar() == 'n' )
           break;

    } while ( 1 );
```

> Eu escolhi o `do..while` porque o código sempre vai ser executado pelo menos uma vez.

E o código final ficou assim:

```
  #include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char *argv[])
    {
      int n1, n2;    // números que iremos ler
      char operacao; // operação
      int res;       // resposta

      do {
```

```
        // lendo dados
        printf("Digite 2 números: ");
        scanf("%i %i", &n1, &n2);

        printf("Você quer somar (+), subtrair (-), multiplicar (*) ou dividir (/)? ");
        scanf("%c %c", &operacao, &operacao);

        // processando dados
        if (operacao == '+')
          res = n1 + n2;
        else if (operacao == '-')
          res = n1 - n2;
        else if (operacao == '/')
          res = n1 / n2;
        else res = n1 * n2;

        // exibindo dados
        printf("\n%i %c %i = %i\n", n1, operacao, n2, res);

        // reiniciando ou interrompendo programa
        printf("Deseja calcular de novo? [S/n] ");
        getchar();
        if ( getchar() == 'n' )
          break;

      } while ( 1 );
      return 0;
    }
```

Desafio 6

Faça um programa que leia 5 números e retorne a soma entre os 3 menores divididos pelo maior.

> Saída:

```
digite 5 numeros: 10 36 88 89 43

    (10+36+43)/89 = 1.0
```

Resposta

Primeiramente precisamos ler os dados

```
int n[5];
    printf("digite 5 numeros: ");
    scanf("%i%i%i%i%i", &n[0], &n[1], &n[2], &n[3], &n[4]);
```

Agora vamos pegar o maior número digitado (já que é mais simples que pegar os 3 menores).

```
int maior = 0;
    for (int i=0; i<5; i++){
      maior = (n[i] > n[maior])?i:maior;
    }
```

Agora a parte mais complidada, existem duas formas de fazer isso, a primeira é comparar manualmente e atrib
variáveis.

```
int menor1 = maior, menor2 = maior, menor3 = maior;
    maior = n[maior];
```

> temos que dar o `maior` como valor, porque para comparar temos que ter certeza de que pode existir um va
> menor, caso atribuamos o `0` a variável sempre vai ter o menor valor

```
for (int i=0; i++; i<5){
      if (n[i] < n[menor1])
        menor1 = n[i];
    maior = n[maior]
    }

    for (int i=0; i++; i<5){
      if (n[i] < n[menor2] && i != menor1)
        menor1 = n[i];
    }

    for (int i=0; i++; i<5){
      if (n[i] < n[menor3] && i != menor2 && i != menor1)
        menor1 = n[i];
    }
```

Essa forma com certeza funciona, mas existe uma forma mais inteligente de fazer:

```
maior = n[maior];
    int menores [] = {maior, maior, maior};
    for (int j=0; j<3; j++){
      for (int i=0; i<5; i++){
        if (n[i]<menores[j]){

          menores[j] = n[i];
          n[i] = maior;

        }
      }
    }
```

E por fim vamos exibir os resultados:

```
printf("(%i+%i+%i)/%i = %1.1f\n",
        menores[0],
```

```
        menores[1],
        menores[2],
        maior,
        (float)(menores[0]+menores[1]+menores[2])/maior
        );
```

Código final:

```
#include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char *argv[])
    {

      // criando e lendo dados
      int n[5];
      printf("digite 5 numeros: ");
      scanf("%i%i%i%i%i", &n[0], &n[1], &n[2], &n[3], &n[4]);

      // pegando o maior valor
      int maior = 0;
      for (int i=0; i<5; i++)
        maior = (n[i] > n[maior])?i:maior;

      // pegando os menores valores
      maior = n[maior];
      int menores [] = {maior, maior, maior};
      for (int j=0; j<3; j++){
        for (int i=0; i<5; i++){

          // caso o numero atual for menor que o menor
          if (n[i]<menores[j]){
            menores[j] = n[i];

            /* o numero atual é o menor de todos
             * e por isso tem que deixar de ser,
             * ou não haverá como pegar o segundo
             * menor e muito menos o terceiro já
             * que ele seria o menor.
             */
            n[i] = maior;
          }

        }

      }

      printf("(%i+%i+%i)/%i = %1.1f\n",
          menores[0],
          menores[1],
          menores[2],
          maior,
          (float)(menores[0]+menores[1]+menores[2])/maior
          );

      return 0;
    }
```

Desafio 7

Faça um algoritmo que leia números inteiros indefinidamente e só pare quando o valor lido for maior que 1000, resultados devem ser informados o maior, o menor, e a media entre eles, além de dizer quais foram repetidos repetidos o número de vezes que foi repetido, além de todos os números primos da lista.

```
digitte números... (para parar digite um numero >= 1000)
    9
    8
    4
    2
    9
    8
    11
    390
    23
    42
    13
    1000
```

Algumas funções e bibliotecas úteis

Até agora só usamos duas bibliotecas em nossos porgramas em C, e não vimos nem 10% dessas, então para de seu conhecimento mais completo aqui vamos mostrar algumas funções dessas duas bibliotecas, além de outras também podem ser bem interessantes.

```
#include <stdio.h>
```

Como já vimos as funções `scanf`, `printf`, `putchar`, `puts`, `getchar`, `gets`, `fprintf` e `fgets`, iremos ignorá-las.

O `std` significa exclusivamente “standard” (“padrão” em português), `i` é de “input” (entrada) e o `o` de “output” (sa portanto entrada e saída de dados padrão.

I/O em arquivos

Como o `stdio` serve para entrada e saída de dados, óbviamente também é usada para manipulação de arqu usada tanto para ler (input), quanto para escrever (output) neles.

Para ler um arquivo precisamos criar um ponteiro do tipo `FILE`

```
FILE * arquivo;
```

Leitura

> crie um arquivo `j.txt` com “joao” escrito dentro

Para abrir o arquivo `j.txt` no nosso programa, é só usar a função `fopen` .

```
arquivo = fopen("j.txt", "r");
```

> O `"r"` no segundo parametro é o modo desse arquivo, nesse caso, abrimos um arquivo em modo leitura.

Para ler e exibir o que foi lido é só usar o `fgetc` :

```
char caractere = fgetc(arquivo);
```

> O `fgetc` retorna um caractere de cada vez, e quando le o caractere, quando você for ler de novo ele lerá somer
> o próximo, exemplo:

```
// suponha que o arquivo "texto.txt" tem "abc" escrito dentro
    FILE *f = fopen("texto.txt", "r");
    char a, b, c;

    b = fgetc(f); // -> "a"
    c = fgetc(f); // -> "b"
    a = fgetc(f); // -> "c"

    print("%c %c %c\n", a, b, c);
```

> Saída:

```
c a b
```

Lembre-se que o fim de um arquivo é demarcado por uma constante chamada de `EOF` (significa “end of file” ou “fir
arquivo”), logo, se voce usar um loop, para ler o tal arquivo, usem o `EOF` como “flag”

> “flag” é a condição de interrupção

```
char caractere;
    do {
      caractere = fgetc(arquivo); // -> <caractere> = fgetc(<arquivo>)
      putchar(caractere);
    } while (caractere != EOF);
```

E depois de terminar de usar, assim como você tem que liberar a memória com o `free` trabalhando com ponte
você tem que fechar o arquivo, ou ele vai ficar ocupando memória à toa

```
fclose(arquivo);
```

E ficaria assim:

```
FILE * arquivo;
    arquivo = fopen("j.txt", "r");

    char caractere;
    do {
      caractere = fgetc(arquivo);
      putchar(caractere);
    } while (caractere != EOF);
    fclose(arquivo);
```

Na minha humilde opnião é muito melhor ler os dados caractere por caractere porque assim se tem mais controle
dados, mas existem outras funções que auxiliam nisso.

O `fscanf` lê dados do arquivo e joga na variável estipulada:

```
FILE *arquivo = fopen("texto.txt", "r");

    char * texto_do_arquivo = malloc(10);
    fscanf(arquivo, "%s", texto_do_arquivo);

    printf ("%s", texto_do_arquivo);

    fclose(arquivo);
    free(texto_do_arquivo);
```

> Não usei a forma de array aqui porque aparentemente o `fscanf` crachou comigo, mas sinta-se a vontade pa
> testar se isso acontece com você também…

O `fscanf` vai ler até o primeiro espaço ou a primeira quebra de linha (`\n`).

Também dá pra fazer com o já conhecido `fgets` :

```
FILE *f = fopen("f.txt", "r");

    char str [100];
    fgets (f, 100, str); // -> fgets ( <arquivo>, <tamanho da string>, <string> )

    fclose(f);
```

E por fim com o `fread` , que é uma forma mais direta de ler os dados, mas para usar o `fread` você tem que sab
limite da leitura (que no nosso caso é o fim do arquivo), e para descobrir isso, nós vamos usar duas funções chama

`fseek` e `ftell`.

```
fseek(arquivo, 0, SEEK_END); // mudando o "cursor" para o fim do arquivo
    size_t tamanho_arquivo = ftell(arquivo); // pegando a posição do cursor
    fseek(arquivo, 0, SEEK_SET); // colocando o cursor no inicio de novo
```

Feito isso é só ler usando o `fread`

```
fread(      texto,        sizeof (char),              tamanho_arquivo, arquivo );
    //   ( <ponteiro>, <tmh do tipo do ptr>, <limite da leitura em bytes>, <arquivo> )
```

E vai ficar assim:

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

    fseek(arquivo, 0, SEEK_END); // mudando o "cursor" para o fim do arquivo
    size_t tamanho_arquivo = ftell(arquivo); // pegando a posição do cursor
    fseek(arquivo, 0, SEEK_SET); // colocando o cursor no inicio de novo

    char * texto = malloc( tamanho_arquivo );

    fread( texto, sizeof (char), tamanho_arquivo, arquivo );

    free(texto);
    fclose(arquivo);
```

> O conteúdo do arquivo vai ser escrito na variável `texto`

Escrita

Para abrir um arquivo em modo escrita ao invés de colocar o `r` no parametro do `fopen`, colocamos um `w`:

```
FILE *arquivo = fopen("texto.txt", "w");
```

> Nesse caso, se o arquivo não existir, ele será criado, mas se existir um arquivo ele perderá todos os seus dados.

Para escrever um `char` em um arquivo usamos a função `fputc`

```
fputc('a', arquivo); // -> fputc( <char>, <arquivo> )
```

Para escrever uma string use o `fputs`

```
fputs("string com coisas", arquivo); // -> fputs(<string>, <arquivo>)
```

Ou se quiser escrever um dados formatado use o já estudado `fprintf`

```
fprintf ( arquivo, "%i > %i = %s", 4, 3, (4 > 3? "True": "False"));
```

E por fim você pode usar o irmão do `fread` , o `fwrite` :

```
char texto [] = "texto aleatorio para colocar no arquivo";
    fwrite( texto, sizeof (char), sizeof(texto), arquivo );
```

Mas esses não são os únicos modos de abertura de um arquivo

```
"r"  // -> read: somente leitura
    "w"  // -> write: somente escrita, mas apaga o conteúdo do arquivo antes de escrever
    "a"  // -> append: somente escrita
    "r+" // -> read/write: leitura e escrita
    "w+" // -> read/write: leitura e escrita, mas apaga o conteúdo do arquivo antes de escrever
    "a+" // -> read/append: leitura e escrita
```

> Faça seus testes com cada um deles, para ver funcionando na prática

Você já deve ter percebido que as mesmas funções que usamos em arquivos, são usadas na `stdout` , `stdi` `stderr` , não é mesmo? Isuuso acontece, porque essas 3 variáveis são arquivos, e por esse motivo você pode todas as funções usadas em arquivos colocando elas no lugar, mas lembre-se, o `stdout` e o `stderr` estão em m `"w"` , enquanto o `stdin` está em modo `"r"` , portanto, você só pode escrever no `stdout` e no `stderr` , e só pode `stdin` . Divirta-se!

Posicionamento em arquivos

Para “finalizar” este assunto, existem algumas funções que podem ser úteis na manipulação de arquivos, com `fseek` (que já foi visto de maneira superficial)

> Não vamos finalizar totalmente porque ainda faltam algumas funções, que agente vai ver no capítulo de `stdarg.h`

```
FILE *j = fopen("j.txt", "r");

    fseek( j, 0, SEEK_SET); // passa o cursor para o inicio do arquivo
    fseek( j, 0, SEEK_CUR); /* passa o cursor para a posição atual do ponteiro
                               se ja tiver lido 3 caracteres, o cursor volta para
                               o caractere 3
                            */
    fseek( j, 0, SEEK_END);
```

Caso queira retornar para o inicio do arquivo, você pode usar a versão simplificada do `fseek` que se chama `rewind`

```
FILE * arquivo = open("j.txt", "r");
    rewind( arquivo );
```

A função `ftell` , também já vista retorna a posição atual do cursor

```
FILE *j = fopen("j.txt", "r");

    char c;
    while ((c = fgetc(j)) != 'a')
      putchar(c);

    printf("\n%li\n", ftell(j));
```

Mas caso você precise de mais controle nesse posicionamento é só usar as funções `fgetpos` e `fsetpos`

```
// j.txt -> "abcdefghijklmnop"
    FILE * arquivo = fopen("j.txt", "r+");

    fpos_t posicao; // tem que ser deste tipo para funcionar

    fgetpos(arquivo, &posicao); // pegando a posição
    printf("posicao: %p\ncaractere: %c", &posicao, fgetc(arquivo));
    fseek( arquivo, 0, SEEK_SET );

    fsetpos(arquivo, &posicao + 4); // mudando posição para o 4 caractere
    fgetpos(arquivo, &posicao);     // pegando a posição de novo

    getchar();

    printf("posicao: %p\ncaractere: %c\n", &posicao, fgetc(arquivo));
    fclose ( arquivo );

    getchar();
```

> Saída:
>
> ```
> posicao: 0x7ffddf294270
>    caractere: a
>    posicao: 0x7ffddf294270
>    caractere: e
> ```

Operações com arquivos

Para apagar o arquivo é só usar a função `remove`

```
remove("j.txt");
```

E para renomear é só usar `rename`

```
rename(       "j.txt",  "joao.txt");
    //    ( <nome antigo>, <nome novo>)
```

A função `reopen` é muito útil para mudar o destino de arquivos, exemplo:

```
freopen("j.txt", "w", stdout);
    fprintf("joao é uma pessoa!!\n", stdout); // o resultado não será impresso na tela, mas no arquivo "j.tx
```

Além de todas essas, lembra de quando imprimimos mensagens na saída de erro (`stderr`) com `fprintf` ? na `stdi` existe uma que faz isso automaticamente; é o `perror`

```
perror("ferrou!!");
```

Para outras informações sobre a biblioteca veja a referência que está no wikibooks sobre ela.

```
#include <stdlib.h>
```

A `stdlib.h` é com certeza uma das bibliotecas mais importantes do C, portanto, merece ser dicecada aqui.

E as funções que já vimos dela foram as de gerenciamento de memória (`malloc` , `free` , `realloc`), logo, não ire revê-las.

Conversões entre string e outros tipos

```
double     d = atof ("8.9"); // atof(<valor>): de string para double
    int        i = atoi ("89");  // atoi(<valor>): de string para inteiro
    long       l = atol ("999"); // atol(<valor>): de string para long
    long long ll = atoll ("99"); // atoll(<valor>): de string para long long
```

Sistema

Caso queira abortar o programa, você pode usar a função `exit` , e assim como no `return` do main, você escol valor que quer retornar para o SO

```
int i;

    scanf("%i", &i);
    if (i%2)
      exit(0); // se for impar saia
    else
```

```
    exit(1); // senao saia e retorne um erro
```

Outra função relacionada ao fechamento do programa é a função `atexit`, que registra funções que serão executa quando o programa finalizar, sendo que estas funções não podem retornar valores e nem receber parametros.

```
#include <stdio.h>
    #include <stdlib.h>

    void tmp_file_remove (void){
      remove("/tmp/at_exit_lock");
    }

    int main(){
      FILE * tmp = fopen("/tmp/at_exit_lock", "w");
      atexit(tmp_file_remove);

      // pausando a execução
      puts("não click em enter ainda... olhe se há um arquivo \"at_exit_lock\" na pasta /tmp/");
      getchar();

      puts("agora veja se ainda está lá");
      fclose (tmp);
      return 0;
    }
```

Outra semelhante à `atexit` é a `at_quick_exit`, que vai ser executada quando o programa for interromido usan função `quick_exit`

```
#include <stdio.h>
    #include <stdlib.h>

    void tmp_file_remove (void){
      remove("/tmp/at_exit_lock");
    }

    int main(){
      FILE * tmp = fopen("/tmp/at_exit_lock", "w");
      atexit(tmp_file_remove);

      // pausando a execução
      puts("não click em enter ainda... olhe se há um arquivo \"at_exit_lock\" na pasta /tmp/");
      getchar();

      fclose (tmp);
      quick_exit(0);

      // essa parte não vai executar
      puts("agora veja se ainda está lá");
      return 0;
    }
```

Outra variável de sistema muito útil é a `getenv`, que retorna o valor de uma variável de ambiente.

```
char path = getenv("PATH"); // caminhos para executáveis no linux ($PATH)
```

E as mais úteis de todas, com essas você vai conseguir executar comandos do sistema operacional

```
system( "echo hello mundo!" ); // system( <comando> )
```

Mas a `system` executa e manda o resultado para a `stdout`, se você quiser acessar o valor de retorno, tem que us função `popen` (que retorna um `stream`, logo, você vai ter que tratá-la como um arquivo)

> A função popen não funciona no C99, se seu compilador usa C99, não irá compilar.

```
FILE *response = popen("echo hello mundo!", "r"); // popen( <comando> )

    char comando [20];
    fgets(comando, 20, response);
    printf("o a resposta do comando usado foi:\n%s\n", comando);
```

Ainda faltam algumas funções mas essas são as mais importantes (contando com as de alocamanto de memória), outras informações sobre a biblioteca, consulte a referência feita por alguém na wikipedia.

```
#include <math.h>
```

Com certeza toda linguagem que se presa tem uma biblioteca de matemática, a `math.h` tem diversas funções resolução de problemas matemáticos desde arredondamento até trigonométricos.

Funções de arredondamento

Digamos que o valor de uma operação dê `1.7`, se quisermos arredondá-lo para cima usamos a função `ceil`:

```
printf("%f\n", ceil(1.7));
```

Mas se quisermos arredondá-lo para um número menor usamos a função `floor`:

```
printf("%f\n", floor(1.7));
```

E se quiser apenas cortar a parte decimal use o `trunc`:

```
printf("%f\n", trunk(1.7));
```

Outra opção é arredondar para o número inteiro mais próximo, seja ele acima ou abaixo:

```
printf("%f\n", round(1.7));
```

A função `round` tem algumas variações como o `lround` que arredonda para um `long int` e o `llround` que arred para um `long long int` .

Potencia e radiciação

Para realizar uma potenciação é só usar a função `pow`

```
printf("40 ao quadrado é %.0f", pow(40, 2));
```

E caso queira fazer uma raiz quadrada é só usar a função `sqrt`

```
float n = pow(40, 2);
    printf("a raiz quadrada de %.0f é %.0f", n, sqrt(n) );
```

E raiz cúbica é `cbrt`

```
float n = pow(40, 3);
    printf("a raiz quadrada de %.0f é %.0f", n, cbrt(n) );
```

E caso você queira fazer uma raiz de índice `5` , `4` ou qualquer outro número, lemre-se que uma radiciação é ap uma potencia elevada à um expoente “ao contrário”:

> 2 normal é igual a 2/1, 2 ao contrário é igual a 1/2

```
// vou usar o expoente 2 mas funciona com qualquer valor
    int numero = pow(5, 2);                  // 25
    int outro_numero = pow(numero, 1.0/2.0); // 5

    printf("5²  = %i\n√25 = %i\n", numero, outro_numero);
```

> E não esqueça dos `.0` após o número se você não fizer isso o valor do expoente vai ser um inteiro e portan será `0` , lunca se esqueça de checar os tipos primitivos…

E existe uma rotina exclusiva para cálculo de hipotenusa:

```
int cateto_oposto = 8, cateto_adjacente = 6;
    int hipotenusa = hypot( cateto_oposto, cateto_adjacente );
```

A biblioteca de matemática tem diversas outras funções, logo, caso necessite fazer algoritmos matemáticos consu
referência da math.h feita pela UFRGS

```
#include <stdarg.h>
```

A `stdarg.h` é uma biblioteca para tratamento de argumentos (ou parametros) de funções.

Até aqui você deve está se perguntando, *"como fazer funções como o `printf` ou o `scanf` que recebem um núr indeterminado de argumentos?"*, exatamente usando esta biblioteca, mas preste atenção para entender como v pode usá-la em seus algorítmos.

Para essa biblioteca, vou explicar de uma maneira diferente, aqui nós vamos criar o `print` , que assim como o `pri` irá esquever coisas na tela.

> Como vai ser a chamada do `print`

```
// print ( <formato>, <dados> );

    print( "isfsf", 90, " + ", 8.3, " = ", 90.0 + 8.3 );

    // i -> %i/%d/%li
    // s -> %s
    // f -> %f/%lf
```

Na declaração da função tem que ter pelo menos 1 argumento fixo, e no nosso caso é o `formato` , todos os ou argumentos serão substituídos por um `...`

```
void print( char * formato, ... );
```

Para acessar os dados no `...` nós primeiro temos que guardar eles em uma variável do tipo `va_list`

```
void print( char * formato, ... ){
      va_list argumentos;
    }
```

Esse va_list é um ponteiro com todos os argumentos, mas para pegarmos os certos temos que dizer para ele de c começar a pesquisar usando o `va_start`

```
void print( char * formato, ... ){
      va_list argumentos;
      va_start( argumentos, formato );
    }
```

Agora iremos checar quantos dados estamos esperando, e depois pegar-los com a função `va_arg`

> Caso for usar valores em `char`, na hora de usar o `va_arg`, usem com `int`, ele não aceita `char` porque é mu pequeno.

```
#include <string.h> // -> strlen
    void print( char * formato, ... ){
      va_list argumentos;
      va_start( argumentos, formato );

      int argc = strlen(formato); // pegando a qntd de caracteres da string

      for (int i = 0; i<argc; i ++){
        if (formato[ i ] == 'i') // caso o dado esperado for um int
          printf( "%li", va_arg( argumentos,      long int ) );
          //             va_arg( <lista de args>,   <tipo> ) );

        if (formato[ i ] == 'f') // float
          printf( "%lf", va_arg( argumentos, double ) );

        if (formato[ i ] == 's') // string
          printf(va_arg( argumentos, char *));
      }
      putchar('\n');
      va_end( argumentos ); // fechando os argumentos
    }
```

> O `va_arg`, assim como o `fgetf`, retorna o dado e passa para o próximo automaticamente.

E fim, essas são as únicas funções que existem nessa biblioteca. Mas como eu prometi no capítulo sobre `stdi` agora eu irei explicar sobre as funções que usam o `va_list` da `stdarg.h`

As funções do `stdio.h` que usam `va_list` fazem o mesmo que as outras, só que aceitam esse tipo de argume como o `vprintf`

```
void escreva_numeros ( int qntd, ... ){
      va_list args;
      va_start( args, qntd );

      char * formato = malloc( qnt*3+1 );
      for (int i=0; i<qntd; i+=2){
        formato[i] = '%';
        formato[i+1] = 'i';
        formato[i+2] = 'i';
      }
      formato[ qnt*3 ] = '\n';

      vprintf(formato, args);
```

```
      va_end( args );
      free(formato);
    }
```

E funciona da mesma maneira com as funções `vscanf` (`scanf`), `vsscanf` (`sscanf`), `vfscanf` (`fscanf`) ...

```
#include <string.h>
```

Esta é mais uma das bibliotecas que eu já falei, mas não me aprofundei, portanto irei ignorar as funçõe mencionadas (`strlen` , `strcpy`).

A primeira função interessante é a `strncpy` , que ao invés de copiar a string inteira, copia apenas um númer caracteres

```
char str[10];
    strncpy(str, "joao e maria", 4); // copia até o 4 caractere
    str[4] = '\0';                   // setando o fim da string

    puts(str);
```

> Saída:

```
joao
```

Outra que também é bacana é a `strcat` , que serve para concatenar strings

```
char str[] = "joao";
    strcat(str, " e maria"); // strcat( <destino>, <destinatario> );
```

E existe a variação `strncat` , que concatena até um certo número de caracteres

```
char str[] = "joao";
    strncat(str, " e maria rosa", 8);
```

Uma função muito útil dessa biblioteca é a `strcmp` que compara duas strings

```
char str [] = "joao", str2 [] = "maria";
    int res = strcmp( str, str2 ); // strcmp( <str>, <str2> )
```

```
    if ( res == 0 )
      puts("as strings são iguais");
    else if ( res < 0 )
      puts("\"%s\" é menor que \"%s\"", str, str2);
    else if ( res > 0 )
      puts("\"%s\" é maior que \"%s\"", str, str2);
```

E também existe a `strncmp` que funciona da mesma forma que a anterior, mas compara só até um certo caractere.

```
char str [] = "joao", str2 [] = "joao e maria";
    int res = strncmp( str, str2, 4 ); // strncmp( <str>, <str2>, <numero> )

    if (res == 0)
      puts("os primeiros 4 caracteres da string 2 são iguais aos da string 1");
```

Outras opções são usar funções de pesquisa em strings, como o `strchr` que irá retornar a string da primeira ocorre de um caractere até o seu fim

```
char j[] = "abcdefghijklmnop";
    puts(strchr(j, 'g'));
```

> Saída:

```
ghijklmnop
```

Outra bem bacana é a `strcspn` onde você passa uma certa lista de caracteres e ela irá retornar a primeira ocorrenc

```
char str[] = "bcdefgh";
    printf("a primeira vogal de \"%s\" está na %i posição\n",str, strcspn(str, "aeiou")+1);
```

Uma semelhante a `strchr` é a `strstr` , que retorna a string da primeira ocorrencia de um caractere até o seu fim

```
char str[] = "joao maria ronaldo";
    puts(strstr(str, "maria"));
```

```
#include <ctype.h>
```

Esta biblioteca possui funções para reconhecimento de tipos de caractere (`char`), e carrega diversas funções para reconhecimento.

```
isnum    ('2'); // se é numerico
    isalpha  ('s'); // se é alfabético
    isblank ('\t'); // se é em branco
    iscntrl ('\n'); // se é caractere especial
    isdigit  ('4'); // se é numero decimal
    isgraph  ('!'); // se tem representação gráfica
    isprint  ('2'); // se dá para escrever na tela
    ispunct  ('.'); // se é pontuação
    isspace ('\v'); // se é um espaço branco
    isxdigit ('0'); // se é hexadecimal
    islower  ('a'); // se é letra minúscula
    isupper  ('A'); // se é letra maiúscula

    tolower  ('A'); // transforma em letra minúscula
    toupper  ('a'); // transforma em letra maiúscula
```

É necessário que eu esclareça alguns pontos, caracteres como □ ou `ç` são grandes demais para caber em um `c` caso precise desses caracteres unicode, use strings para representá-los.

Considerações finais

Espero que este livro tenha ajudado você, este não é o fim definitivo, os seus estudos não devem acabar aqui, e nã preocupe pois vem aí o próximo livro falando sobre criação de interfaces gráficas e manipulação de imagens, vid músicas, etc. Então tenha paciência e antes de ir embora vou da um bônus para você se animar e começar s projetos em C.

Criando um projeto em C

Vamos utilizar o projeto do pantuza, que se chama c-project-template, que faz toda a estrutura automaticamente.

> Para instalar basta clonar o repositório usando o `git`

```
git clone https://github.com/pantuza/c-project-template.git
```

Essa é a arvore de diretórios do `c-project-template` :

```
.
    ├── bin
    ├── lib
    ├── LICENSE
    ├── log
    ├── Makefile
    ├── project.conf
    ├── README.md
    ├── src
    │   ├── args.c
    │   ├── args.h
    │   ├── colors.h
```

```
│   ├── main.c
│   ├── messages.c
│   └── messages.h
└── test
    └── main.c
```

E para configurar o seu projeto você só precisa editar o arquivo `project.conf`, que devará estar mais ou menos ass

```
#
# This is a project configuration file. Change variables to generate
# your project correctly
#

#
# Put here your project name. It will create a root directory with this name
#

PROJECT_NAME := project


#
# Put here the binary file name. The compilation will result in that binary
#
BINARY := binary


#
# The project directory. At this path we will start the project
#
PROJECT_PATH := ~/projects/$(PROJECT_NAME)
```

E para exemplificar, o nosso `Hello mundo` que criamos no inicio do livro vai ser nossa cobaia:

Dentro deste arquivo, você vai colocar no `PROJECT_NAME` o nome do projeto, no `BINARY` o nome do arquivo compilação, e no `PROJECT_PATH` o diretório do projeto que você está criando.

> project.conf:

```
PROJECT_NAME := Hello
    BINARY := hello
    PROJECT_PATH := ~/Create/Projects/Hello
```

Após editar o arquivo rode o comando `make start` dentro do diretório do `c-project-template`

Pronto, agora você pode entrar no diretório do seu projeto e editar os códigos da pasta `src`, e quando for compilar o comando `make`