Apostila

C

Progressivo

www.cprogressivo.net

Sobre o e-book C Progressivo

Antes de mais nada, duas palavrinhas: parabéns e obrigado.

Primeiro, parabéns por querer estudar, aprender, por ir atrás de informação. Isso é cada vez mais raro hoje em dia.

Segundo, obrigado por ter adquirido esse material em www.cprogressivo.net
Mantemos neste endereço um *website* totalmente voltado para o ensino da linguagem de programação C.

O objetivo dele é ser bem completo, ensinando praticamente tudo sobre C, desde o básico, supondo o leitor um total leigo em que se refere a absolutamente tudo sobre computação e programação.

Ele é gratuito, não precisa pagar absolutamente nada. Aliás, precisa nem se cadastrar, muito menos deixar e-mail ou nada disso. É simplesmente acessar e estudar.

E você, ao adquirir esse e-book, está incentivando que continuemos esse trabalho.
Confiram nossos outros trabalhos:
www.programacaoprogressiva.net
www.javaprogressivo.net
www.pythonprogressivo.net
www.htmlprogressivo.net
www.javascriptprogressivo.net
www.excelprogressivo.net
www.assemblyprogressivo.net

Certamente, seu incentivo $$ vai nos motivar a fazer cada vez mais artigos, tutoriais e novos sites!

Este e-book é composto por todo o material do site. Assim, você pode ler no computador, no tablet, no celular, em casa, no trabalho, com ou sem internet, se tornando algo bem mais cômodo.

Além disso, este e-book contém mais coisas, mais textos e principalmente mais exercícios resolvidos, de modo a te oferecer algo mais, de qualidade, por ter pago pelo material.

Aliás, isso não é pagamento, é investimento. Tenho certeza que, no futuro, você vai ganhar 10x mais, por hora trabalhada, graças ao conhecimento que adquiriu aqui.

Proprietário da Apostila

Esse e-book pertence a:

Nome: Empório Mendes
E-mail: emporiomendes1@gmail.com
Telefone: (34) 32278000
CEP: 38402290
Endereço: AVENIDA JOSE DO PATROCINIO
Número: 345
Complemento: Comercio
Bairro: Marta Helena
Cidade: UBERLANDIA
Estado: MG

Pedimos, encarecidamente que não distribua e comercialize seu material. Além de conter suas informações, poderia prejudicar muito nosso projeto.

Se desejar indicar o C Progressivo para um amigo, nosso site possui todo o material, de forma gratuita, sem precisar de cadastro e o acesso dessas pessoas também ajudar a mantermos o site no ar e criamos cada vez mais projetos:

www.cprogressivo.net

Sumário

Para os iniciantes ou que querem aprender bem mesmo, que siga os seguintes tutoriais na ordem que é colocada.

Para ler um tutorial avulso, a gente assume que você tenha lido ou tenha os conhecimentos de todos os artigos anteriores.

Estude no seu próprio tempo, sem pressa, com calma, leia e releia quantas vezes forem necessárias.

Qualquer dúvida, estamos lá no site a disposição para dúvidas e debates.
Se preferir, também temos nosso grupo no Facebook:
Programação Progressiva

Bons estudos!

Tutorial de Conceitos Básicos da Linguagem C

Teste Condicional e Controle de Fluxo

1. O teste condicional IF ELSE
2. Fazendo testes e comparações - operador de igualdade (==), maior (>), menor (<), maior igual (>=), menor igual (<=), de diferença (!=) e de módulo, ou resto da divisão (%)
3. Questões sobre IF ELSE
4. Operadores de Incremento (++), Decremento (--) - Diferença entre a=b++ e a=++b
5. O laço WHILE: o que é, para que serve e como usar
   5.1 Questões sobre o laço WHILE
   5.2 Soluções das questões sobre o laço WHILE
6. O laço FOR: o que é, para que serve e como usar o FOR - Cast
   6.1 Questões sobre o laço FOR
   6.2 Soluções das questões sobre o laço FOR
7. Os comandos CONTINUE e BREAK em C: pausando e alterando o fluxo de laços
8. O teste condicional SWITCH: o que é, para que serve e como usar o switch
9. O laço DO WHILE: o que é, para que serve e como usar o do while
   9.1 Programa em C: Criando uma calculadora usando DO WHILE e SWITCH
10. Exercícios envolvendo testes e laços
11. Solução dos exercícios sobre testes e laços

Funções em C

1. O que são funções, para que servem e como usá-las
   1.1 Programa em C: Criando um chat com menu através de funções
2. Argumentos e Parâmetros de funções - Funções aninhadas
3. O comando return : devolvendo informações para quem invocou as funções
4. Variáveis locais - Protótipo de uma função
5. Gerando números aleatórios em C: rand, srand e seed
   5.1 Jogo em C: Adivinhe o número que o computador sorteou através das dicas!
6. Funções recursivas: pra aprender recursividade, tem que saber recursividade
7. Exercícios sobre funções

7.1 Exercícios resolvidos sobre funções

Estrutura de dados I: Vetores/Arrays

1. O que são vetores, como declarar e quando usar
2. Inicializando vetores - Vetores de caracteres e Lixo
3. Não use números, use constantes: const e #define
4. Matrizes em C: Vetores multidimensionais (Vetor de vetores)
5. Como passar vetores e Matrizes para funções
6. Passagem por Referência - Como copiar Vetores e Matrizes
7. Exercícios sobre Vetores e Matrizes
8. Jogo: Como fazer o Jogo da Velha

Ponteiros (apontadores)

1. Introdução ao uso dos ponteiros: Endereços de memória
2. A função sizeof() e os blocos vizinhos de memória
3. Como declarar, inicializar e usar ponteiros - A constante NULL
4. Variáveis apontadas - A Passagem por Referência
5. Operações matemáticas com Ponteiros

Strings e Caracteres - Escrevendo em C

1. Introdução ao uso das strings: O que são, como declarar, inicializar e o caractere \0
2. Lendo e Escrevendo Strings
3. Como criar uma biblioteca (ou header .h) em C
4. A biblioteca string.h e suas funções
   4.1 Exercício: Implemente as funções da biblioteca string.h
5. Exercícios sobre Strings

Estrutura de dados II: structs

1. Introdução as structs: O que são, para que servem e onde são usadas
2. Como declarar uma struct
3. Como acessar, ler e escrever em elementos de uma struct
4. typedef: Como criar seus próprios tipos
5. Como enviar structs para funções
6. Como passar uma struct para funções por referência - O operador ->
   6.1 Exercícios sobre structs em C

Alocação Dinâmica de Memória

1. Introdução: o que é alocar memória dinamicamente e para quê isso serve
2. A função malloc(): como alocar memória em C
3. A função free(): Como liberar memória em C e evitar vazamento (Memory Leak)
4. A função realloc(): Realocando memória e a função calloc()

Estrutura de dados III: Estruturas Dinâmicas

1. Estrutura dinâmica de dados em C: O que são Listas, Filas, Pilhas e Árvores
2. Listas (List): O que é e como funciona
   2.1 Inserindo nós no início e final da lista
   2.2 Retirando nós do início e final da lista
   2.3 Implementação completa de uma lista - Inserindo e retirando de qualquer local
3. Pilhas em C - Como Programar
4. Filas em C - Como Programar

Arquivos (FILES) em C

1. Introdução: o que são, como funciona e para que servem os arquivos (files)
2. Abrindo arquivos (fopen), modos de abertura (read r, write r, append a) e fechamento (EOF, fclose e fcloseall)
3. Escrevendo em arquivos: As funções fputc, fprintf e fputs
4. Como ler arquivos em C: As funções fgetc, fscanf e fgets

Como começar a programar

Sempre que alguém fala em programador, cientista da computação, engenheiro de software e coisas do tipo, no imaginário popular vem logo a imagem de alguém *nerd*, gênio ou vulgo CDF (como chamamos aqui no nordeste).

Não, não precisa ser um gênio pra isso.
Não precisa tirar só 10 na escola pra aprender a programar
Não precisa ir pra faculdade pra aprender computação

Aliás, sabia que muito dos gênios desse ramo, abandonaram a faculdade?
Muitos donos de empresa de tecnologia, começaram programando, estudando sozinho e hoje são muito bem sucedidos, e alguns só tem ensino médio?

Pois é, se essa galera, antigamente, sem internet, sem Youtube, sem e-books e PDF, aprenderam, não tenha dúvidas: você pode e vai aprender também.

Não precisa ser inteligente nem ter conhecimento prévio algum.
Mas depois de aprender a programar, se prepare: sua mente vai mudar completamente.

O raciocínio muda.
A criatividade muda.
Sua lógica muda.
Sua mente muda totalmente, é algo incrível.

Mas vamos te dar algumas orientações, talvez não goste de algumas, mas te falo com sinceridade e *mando a real na lata*:

1. Você precisa se esforçar. Programar é simples, mas não simplório. Vai precisar ler, reler, pensar, pensar de novo, tentar e tentar. Isso, as vezes, é desgastante.

2. Precisa arranjar tempo. Muitos que querem programa estão na escola, faculdade, outros até trabalham e tem família. Você vai precisar de tempo. Boa parte do que aprendi, foi estudando em ônibus (não tinha *smartphone*, imprimia mesmo os livros e ia lendo no ônibus).

3. Precisa de sacrifícios. As vezes da vontade de ver uma série na Netflix, mas vá estudar. As vezes dá vontade de ficar dando *refresh* no Instagram e não fazer nada, mas estude. Você vê um vídeo no Youtube e ele te indica 20 outros legais, eu sei como é, mas estude. Se você continuar agindo como todos, se esforçando como todos, gastando seu tempo como todos...vai ser como todos. Quer algo diferente? Precisa agir diferente, amigão.

4. Tente de novo. As vezes, você vai ver exercícios aqui que você vai ler, tentar e vai falhar. No começo, vai ser sempre. Outras vezes, vai tentar tentar...e nada. Dá vontade de ver a resposta, a solução, mas tente mais. Dê uma volta, saia, vá passear, comer algo, namorar e depois tente novamente com a cabeça fresca. Programação é isso, todo dia você vai precisar solucionar algo, tem que quebrar a cabeça, até chorar em posição fetal no banheiro. FAZ PARTE. Só assim se vira um bom programador, ok? Tentando de novo.

5. Você vai se frustrar. Você vai passar 1 mês em um projeto, vai suar, quebrar a cabeça e resolver tudo em 5 mil linhas. Depois descobre que um filho de uma mãe resolveu de maneira melhor, mais rápida, completa e eficiente em 500 linhas. Você ficar pistola, com vontade de desistir e com vontade de decorar uns livros de leis e artigos pra passar num concurso e nunca mais ter que estudar e quebrar a cabeça. Mas se parar pra estudar, ver a solução e como funciona a mente dos outros, você vai ficar cada vez mais e mais fodástico.

6. Você vai sempre precisar estudar. Não importa o quanto estude, vai sempre precisar estudar. Sempre tem uma coisa nova pra aprender, algo que ainda não sabe, sempre vai precisar ler uma documentação de alguma API ou alguma dúvida em algum fórum. Faz parte, tem ter que gana e vontade de aprender. Não se admire se estiver trabalhando programando em um empresa e quando chegar em casa tu do que vai querer é...programar naquele seu projeto pessoal.

7. Estude inglês. Não sabe? Comece. Traduza umas músicas que gosta, veja seriados com legendas em inglês e voz em português, depois reveja com áudio original e a legenda em pt-br. Se puder, faça um curso. Inglês é a língua universal, programadores brasileiros, chineses, indianos, africanos, alemães etc etc, falam em inglês, até os americanos falam em inglês. É MUITO IMPORTANTE ESTUDAR INGLÊS. E calma, não precisa se apressar, dá pra aprender aos poucos, recomendo o Curso de Inglês da Brava Cursos. Clique aqui para entender melhor a importância do inglês no estudo da programação.

No mais, é sem mistério. Senta essa bunda aí na cadeira, e estuda, estuda, tenta, programa, estuda, pesquisa...e se precisar, manda sua dúvida:

programacacao.progressiva@gmail.com

Por favor, qualquer erro de código, lógica, português ou uma solução que achar melhor, nos avise. Impossível não fazer um material desses sem erros. Nos ajude a melhorar cada vez mais o material.

Básico

O necessário para programar em C

- **Ferramentas Necessárias para programar em C**

- Compilador
- Debugger
- Editor de texto

Você vai escrever seus códigos de programação C em qualquer editor de texto e vai usar o compilador.
O compilador converte seu código para código de máquina (um código que só a máquina entende, para rodar no seu computador - o famoso binário) e o debugger faz o debugging, ou seja, checa se há erros no seu código.

Porém, fazer isso tudo manualmente dá muito trabalho. Existe um tipo de programa que faz isso tudo sozinho.

É a IDE, *Integrated Development Enviroment*, ou seja, o ambiente de desenvolvimento integrado.

A título de informação, vamos apresentar três IDE's, o Dev-C++, Visual Studio e o Code::Blocks, porém, **aconselhamos o uso do Code Blocks para iniciantes.**

- **Que programa escolher para programar em C**

  - **Dev-C++: desatualizado e com muitos erros**

Este é o mais usado e indicado nas faculdades e na Internet. Mas se é o mais usado e indicado, por que o curso C Progressivo não indica?

Porque ele é obsoleto! Ele costumava ser bom, e por isso era muito indicado.

MAS ELE PAROU DE SER DESENVOLVIDO!
O PROJETO DO DEV-C++ FOI ABANDONADO!

Mas continuaram a usar e recomendar, principalmente para iniciantes.

Porém, conforme você for avançando, ele ficará nitidamente ruim e desatualizado, irá prejudicar MUITO você!
Infelizmente, seu debugger é cheio de erros! Você poderá se prejudicar caso erre e o Dev-cpp não te alerte sobre os erros.

- **Microsoft Visual Studio: bom, poderoso e pago**

O Visual Studio é tão poderoso que os desenvolvedores da Microsoft fazem o próprio Windows e seus programas/sistemas são feitos usando o Visual Studio. Porém, é da Microsoft. Ou seja, pra usar tudo que a ferramenta tem a oferecer, você tem que pagar - e muito.
A Microsoft, como forma de marketing, porém, lançou uma versão gratuita do Visual Studio, o Visual Studio Express.

Eu, particularmente, acho ele muito pesado para um iniciante. Quem está começando não vai usufruir nem 10% do que ele tem a oferecer, embora tenha baixado centenas MB.
É como matar uma mosca com uma bala de canhão.

Vá com calma. Caso tenha interesse, no futuro, e queira criar aplicações gráficas para Windows (inclusive para o Windows 8), Windows Phone, tecnologia .NET e web, você pode começar a usufruir melhor os recursos dessa poderosa ferramenta de desenvolvimento.

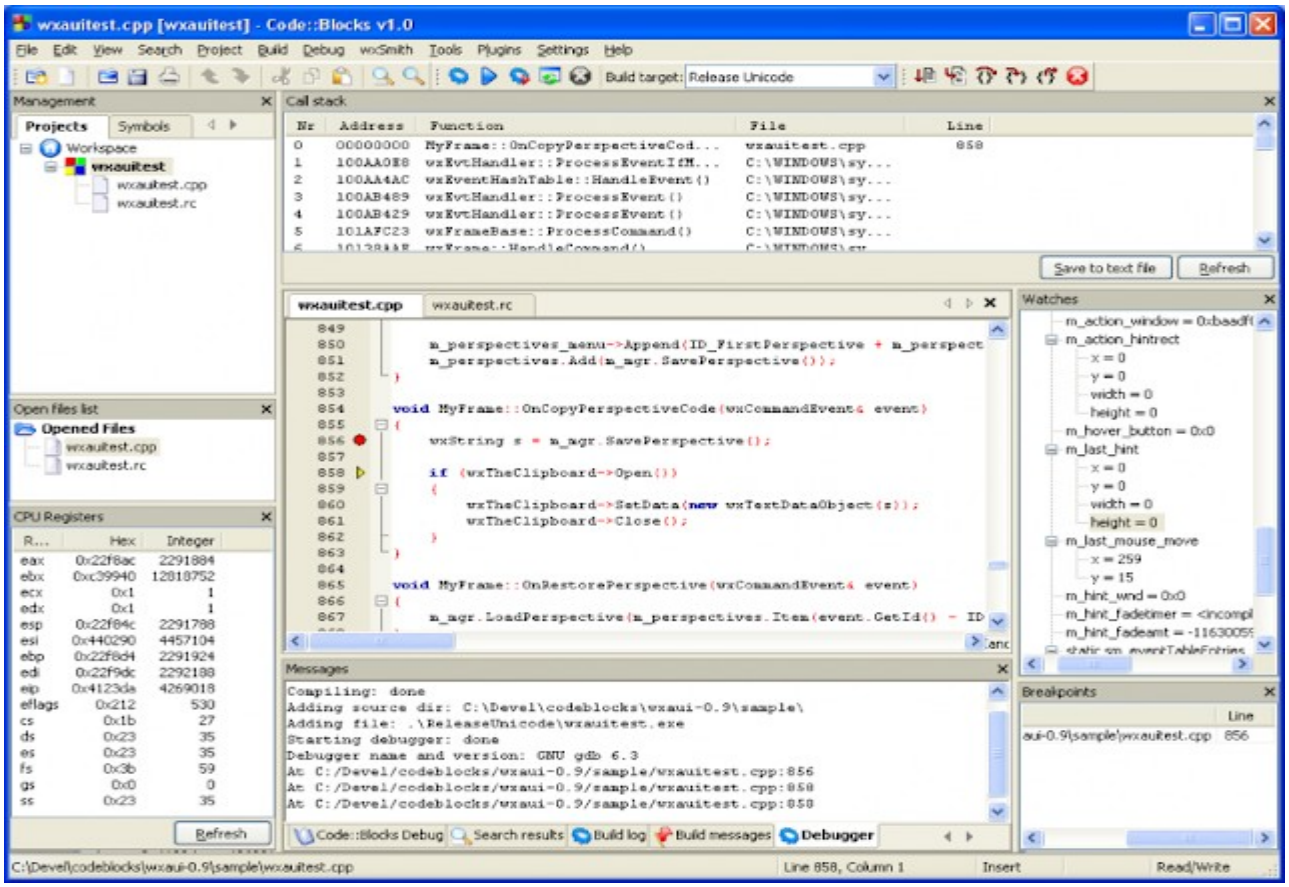
Porém, é sempre bom se informar:
Site do Visual Studio Express
Informações sobre a tecnologia .NET e cursos oferecidos pela Microsoft para seus produtos

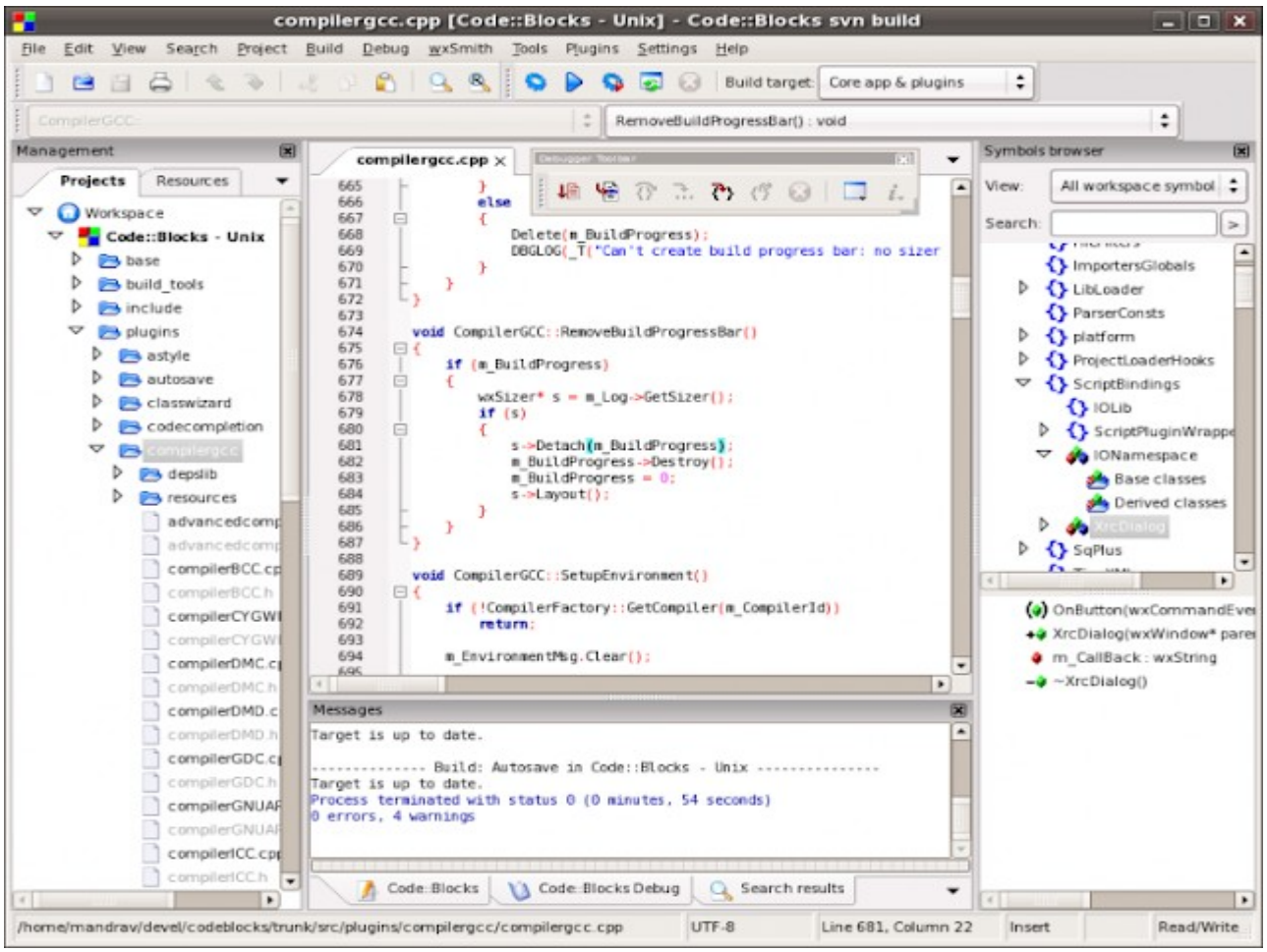
- **Code::Blocks: gratuito, leve, open source e cross plataform**

Veja as razões do Code::Blocks ser melhor e mais recomendado para iniciantes:
- gratuito
- leve
- open source (é possível ver seu código-fonte, como foi feito)
- cross plataform (funciona em várias plataformas, como Windows e Linux)
- está atualizado
- está em desenvolvimento

- é possível expandir suas funcionalidades através dos plugins
- é leve, principalmente se comparado com o Microsoft Visual Studio

Screenshots do Code::Blocks:





- **Porque usaremos o Code::Blocks no curso de Programação C**

Assim como o Code Blocks, o curso de Programação C, *C Progressivo*, é gratuito. Então não apoiaremos a pirataria nem o uso de software pagos. Felizmente, existem milhões de pessoas ao redor do mundo empenhadas em criar ferramentas boas que não deixam a desejar em **absolutamente nada** em relação as pagas.

- **Como começar a programar em C: baixando o necessário para a Apostila C Progressivo**

Instalando o Code::Blocks, você já tem o debugger, compilador e editor de texto.
Isso mesmo. Não precisa baixar mais nada, somente a IDE.
Ao escrevermos o código, o Code Blocks já organiza automaticamente nosso código e quando colocarmos o programa para rodar, ele nos mostrará onde os erros estão.
Caso exista erros, o programa rodará diretamente do Code::Blocks.

Então sem perda de tempo, baixe o programa.
O site do programa é: http://www.codeblocks.org/

Navegue até a seção de download e escolha sua plataforma, Windows, Linux ou Mac OS X, bem como sua versão (Windows 7 ou XP, por exemplo): http://www.codeblocks.org/downloads/26

Como há várias opções, sugerimos que baixe a maior (que tem mais megabytes), pois certamente trará mais recursos e evitará problemas.
Não há segredos na instalação.

E pronto, você já está pronto para começar a programar em C com o curso online de C do site **C Progressivo.**

Seja bem vindo à linguagem de Programação C, a linguagem mais usada do mundo.

Criando e compilando seu primeiro programa na Linguagem C

- **Como criar e compilar um programa em linguagem C**

Passo 1: Inicie um novo arquivo
Dependendo da linguagem em que você instalou o Code::Blocks
Vá em: ***File -> New -> Empty File***
Ou em: ***Arquivo -> Novo -> Arquivo vazio***

Note que apareceu uma tela em branco, que é onde você vai digitar seu código.
Não digite nada ainda. Você até pode, mas é um erro fazer isso, o ideal é salvar o arquivo primeiro.
Vou explicar o motivo no passo 2.

Passo 2: Salve seu arquivo com a extensão .c

O Code::Blocks não serve somente para a linguagem C, serve para a linguagem C++ também.
Como você vai programar em C, seus arquivos devem ter a extensão '.c'.
Clique no disquete, símbolo universal de Salvar e escolha um nome e coloque a extensão .c, por exemplo: **programa1.c**

Após salvo, o Code::Blocks vai indentar automaticamente o seu código, ou seja, vai organizar ele e escrever algumas coisas por você, além de mostrar algumas coisas com cores diferentes, o que facilita e deixa o código mais organizado, **coisa que não aconteceria caso não tivesse salvo antes o programa** (você entenderá melhor isso no próximo passo).

Passo 3: Programando

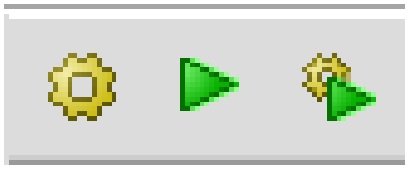
Agora vamos programar! Ou seja, escrever o código!
Escreva EXATAMENTE isso:

```
#include <stdio.h>

int main(void)
{
    printf("Meu primeiro programa - C Progressivo!\n");
    return 0;
}
```

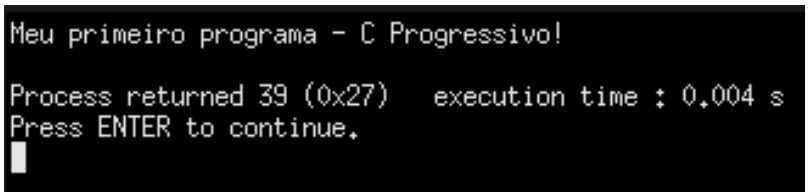
Passo 4: Compilando e Rodando

Note no canto superior esquerdo, os seguintes botões:



O primeiro é o 'Build', o segundo é o 'Run' e o terceiro faz os dois 'Build and Run'.
Clique no terceiro, que ele irá construir (compilar) e executar seu programa, ou aperte F9.

Você obterá a seguinte tela:



E eis o seu programa na Linguagem C.
No próximo artigo explicarei, detalhadamente, o que significa cada parte do código que você escreveu e executou.

Passo 5: Caso tenha obtido erros

Caso não tenha conseguido rodar seu programa, provavelmente deve estar usando outra IDE que não seja o Code::Blocks, então experimente trocar a linha:

main()

Por
int main()

Ou
void main()

Se os erros persistirem, leia a mensagem de erro. Muito provavelmente você digitou algo errado.

Tem que ser exatamente aquilo.
Em C, 'main' é diferente de 'Main'. Ou seja, C é case sensitive.

Código comentado do nosso primeiro programa em C

No artigo passado nós criamos e compilamos nosso primeiro programa em C de nosso curso de programação. Porém, não explicamos exatamente o que aconteceu. Simplesmente mostramos o que escrever e o que fazer.

Em nosso curso você verá sempre, **sempre**, as coisas bem explicadas. Esse é o diferencial do curso online C progressivo.
Veja agora os comentários, linha por linha, do código C.

- **Como criar seu primeiro programa em C**

  Hello world em C

No tutorial passado de nossa apostila de C, ensinamos como criar, compilar e executa seu primeiro programa em C, conhecido como "Hello World" ou "Olá, Mundo!".

E mostramos que seu código é:

```
#include <stdio.h>

int main(void)
{
    printf("Meu primeiro programa - C Progressivo!\n");
    return 0;
}
```

Pois bem, vamos agora explicar o que é isso tudo.

- **O que é e para que serve #include <stdio.h>**

Como já dissemos aqui e na seção *Comece a Programar* do site *Programação Progressiva*, mais especificamente, sobre a Linguagem C, ela é extremamente versátil e poderosa.

O kernel - miolo do sistema - do Linux e do Windows (e da maioria dos sistemas operacionais) é feito, em sua maioria, com a linguagem C.

Programas de alto rendimento, jogos e microcontroladores também podem funcionar sob linguagem C.
O C não está "preparado" ou "no ponto" para você fazer todo e qualquer tipo de aplicação, pois isso o deixaria extremamente lento e pesado, pois teria que incluir muitas funcionalidades.

Quando for programar você vai dizer ao C o que vai fazer e que funções da linguagem C você vai precisar em sua aplicação. E isso é feito, em parte, através da importação de bibliotecas.

Por exemplo, existem bibliotecas gráficas (Allegro, OpenGL etc), que são usadas para criar aplicações gráficas (como jogos ou programas com janelas, botões etc), existem bibliotecas para usar as funções do sistema operacional windows (biblioteca *windows.h),* existem bibliotecas para fazer cálculos matemáticos (*math.h)* e por aí vai.

Não existe um número específico de bibliotecas, e podemos, inclusive, criar as nossas. Então é óbvio que o C não vai carregar todas elas automaticamente. Por isso precisamos dizer ao compilador o que vamos usar.

Para fazer isso, usamos o '#include', que é chamado de diretiva e em seguida escrevemos o nome da biblioteca. No caso, vamos usar a biblioteca '*stdio.h'.*

Dentro dessa biblioteca existe um código em C.
É como se você dissesse ao C: 'ei, adicione esse arquivo: stdio.h no programa, vou precisar das funções contidas nele'.

O **std** de 'stdio' é de *standard*(padrão) e o **io** é de Input/Output (entrada e saída).
Por entrada, entenda o sistema receber dados, como você digitar algo.
Por saída, entenda como um resultado que o sistema te dá, como uma mensagem na tela.

Existem diversos tipos de entrada e saída, mas essa biblioteca trata das entradas e saídas padrões, como o nome diz.

No nosso caso, vamos usar a função *printf* como saída de dados.

- **O que é e para que serve int main(void) { }**

main() é a função principal. Sempre que compilamos um código em C, seu início se dará através dessa função. Tudo começa a partir dela.
O código da função é tudo aquilo que fica entre as chaves: {}
Ou seja, seus códigos em C **sempre devem possuir a função main.**

Nesse nosso primeiro programa, ele simplesmente deve exibir uma mensagem na tela.
Obviamente, o comando para isso deve estar dentro da função main(), senão o código não será executado. Teste. Coloque o *printf* em outro local e veja o Code Blocks apontar esse erro.

- **O que é e para que serve printf("Meu primeiro programa - C Progressivo!\n");**

Print, em inglês, é imprimir. Se acostume com essa notação.

O que nossa função *printf()* faz é imprimir uma mensagem na tela. Essa mensagem ou texto, nós chamamos, em programação, de String. Note que:
String -> "Meu primeiro programa - C Progressivo\n"
Não é uma string -> Meu primeiro programa - C Progressivo\n

Se colocar uma frase sem as aspas duplas, obterá um erro, pois a função *printf* é feita para receber e exibir uma string. Se você não usar as aspas, não estará passando uma string para a *printf*, portanto seu programa irá mostrar um erro.

Note que, só podemos usar a função *printf()* porque importamos a biblioteca *stdio.h*
Essa função está declarada lá nessa biblioteca. O nosso programa final, o executável, também utiliza o código da *printf()*, porém é inútil ficar repetindo código.

Ao invés disso, guardamos nossos códigos para que possamos reutilizar depois.
Essa é uma função das bibliotecas em C.

Mais adiante, em nosso curso online de C, aprenderemos mais sobre como criar nossas próprias funções, bibliotecas, a manusear strings e o printf.

\n: New line, adicionando uma linha em branco

Já o caractere '\n' é o *New Line*, ou seja, ele imprime uma linha.
Ou pula de linha. É como se tivéssemos pressionado enter no terminal de comando, pois faz o cursor saltar uma linha.

Aprenderemos mais sobre o \n nosso tutorial sobre a função printf().

Experimente adicionar mais \n e ver o resultado!

```c
#include <stdio.h>

int main(void)
{
    printf("Meu primeiro programa - C Progressivo!\n\n\n");
    return 0;
}
```

A função printf() e alguns caracteres especiais

Você aprendeu no artigo passado como criar seu primeiro programa na fantástica linguagem de programação C: um aplicativo que exibia uma mensagem na tela.

Pra isso, você usou a função **printf**. Porém, essa função é cheia de recursos, e vamos ensinar mais alguns agora.

Vamos mostrar uma porção de códigos, mas não vamos mostrar o resultado. Para você aprender a programar nunca fique copiando e colando, vá lá e digite os códigos.

Só assim você aprenderá.

- **Caracteres Especiais**

Você viu que para exibirmos uma mensagem, basta escrever o texto entre aspas duplas **""**.

Porém, como vamos ver nesse tutorial de C, não bastante somente escrever as coisas entre aspas duplas, e que tudo vai sair exatamente como você planejou.

Teste, por exemplo, exibir uma mensagem com os seguintes caractere: **"** ou ****

O código ficaria assim:

```
#include <stdio.h>

int main(void)
{
    printf("Aspas duplas "\n");
    printf("Barra: \");
    return 0;
}
```

Como explicamos antes, o que aparece no console (telinha preta) é o que está entre aspas: " isso aqui aparece".

Se você colocar aspas duplas você terá aspas duplas, o C vai interpretar que é pra ser exibido o que estiver entre a primeira e a segunda aspas, que nosso caso é o texto "*Aspas duplas :"*. E a terceira aspas?

A terceira não era pra estar lá, o *n* é um erro, por isso gera um erro na hora de compilar.

- **Por que o C não exibe esses caracteres?**

Porque o nosso conceito de caracteres é diferente do conceito que os computadores tem.
Para nós, o caractere " é aspas duplas. Para o C, é um caractere que delimita o que vai aparecer no console.

Ou seja, pro C, esses caracteres não fazem parte do alfabeto, não estão lá por motivos de leitura para humanos, são símbolos que representam outras coisas.

Então, como exibir esses caracteres especiais na forma de texto?
Usando o caractere de ****
Quando queremos que o caractere de aspas duplas apareça no console, colocamos o \ antes: \"
O C vai interpretar que, com esse símbolo \ antes, o caractere **"** deve ser exibido na tela.

- **Como exibir o caractere** \

Coloque uma barra antes dele também.
Sim, vai ficar: \\
Quando o C encontra essas duas barras, ele entende que deve se exibir o caractere \

Nosso programa fica assim:

```
#include <stdio.h>

int main(void)
{
    printf("Aspas duplas \" \n");
    printf("Barra: \\");
    return 0;
}
```

Em suma, o que esse caractere faz é 'avisar' ao C que o próximo caractere, que vem logo após o **** terá um significado diferente.

Note que já usamos isso antes, para imprimir uma quebra de linha: **\n**
Você sabe que ao escrever *n* o C não irá imprimir a barra nem o **n**, ele vai imprimir uma quebra de linha, ou um 'Enter'.

- **Sinais sonoros e outros caracteres especiais**

Carriage return: \r

Esse caractere especial faz com que o cursor se mova para o início da linha:

```
#include <stdio.h>

int main(void)
{
    printf("Carriage return: \r");
    getchar();
    return 0;
}
```

O comando **getchar()** faz com que o programa espere que o usuário digite alguma tecla.
Assim o programa só termina se você digitar algo. Usamos isso para motrar que o carriage return faz com o cursor vá para o início da linha.

Tabulação horizontal (TAB): \t

```
#include <stdio.h>

int main(void)
{
    printf("Antes do \\t \t Depois do \\t");
    return 0;
}
```

Sons: \7 e **\a**

Os leitores mais antigos, que jogaram video-games como o NES ou Atari certamente vão se lembrar desse aviso sonoro, que também era utilizado em sistemas operacionais antigos e jogos de computador feitos no terminal:

```
#include <stdio.h>
int main(void)
{
    printf("\7 \a");
    return 0;
}
```

Como comentar seus códigos em C - Comentários e Delimitadores

A medida que seus códigos na linguagem C forem aumentando, eles ficarão incrivelmente difíceis de serem entendidos por outra pessoa.

Sim, futuramente seu código será lido/alterado por outra pessoa, provavelmente você.
Para facilitar esse processo, você pode fazer 'comentários' em seus programas de C, explicando o que cada trecho de código faz.

Além de serem importantes, são considerados uma boa prática de programação. Então, se você quer ser um bom programador C, deixe seus códigos comentados e bem explicados.

- **Comentando códigos em C - Usando //**

Sempre que quiser comentar alguma linha de seu código C, inicie a linha com duas barras: //

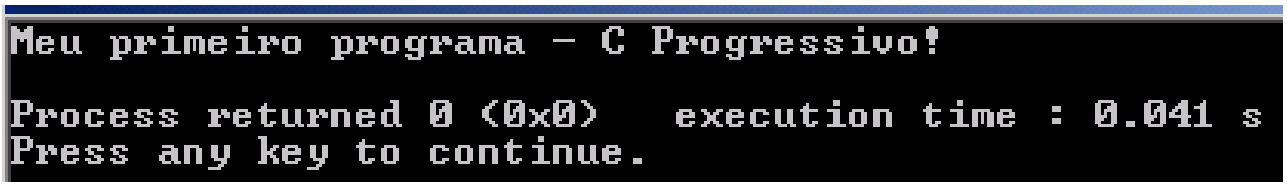
Não importa o que você escreva depois dessas duas linhas, elas não serão vistas.
Ela é bastante usada antes de algum trecho de código C, explicando o que as próximas linhas fazem no programa, evitando assim que quem esteja lendo tenha que quebrar a cabeça para adivinhar o que o programador tentou fazer.

Veja o código exemplo:

```
#include <stdio.h>

int main()
{
    //O seguinte trecho mostra uma mensagem na tela
    //Essas linhas comentadas não irão aparecer na tela
    printf("Meu primeiro programa - C Progressivo!\n");
}
```

O resultado do código compilado é:



No que as linhas que começam com **//** foram totalmente ignoradas.
Você pode colocar uma ou milhão de linhas: na hora de compilar e linkar, o compilador C irá excluir todas as linhas começadas em //
Logo, os comentários servem apenas para a leitura humana, não afetando em absolutamente nada o desempenho de seu aplicativo C.

Outro uso comum dos comentários é na criação de cabeçalhos de seus programas em C, como no exemplo abaixo:

```
#include <stdio.h>
// Código que exibe uma mensagem na tela
// Por Fulano de Tal
// Em 21/12/2112
// Para Curso C Progressivo
int main()
{
   //O seguinte trecho mostra uma mensagem na tela
   //Essas linhas comentadas não irão aparecer na tela
   printf("Meu primeiro programa - C Progressivo!\n");
}
```

- **Como usar os delimitadores /* */ em linguagem de programação C**

Imagine agora que você precisa fazer um comentário de mais de 20 linhas. Isso é bem comum entre estudantes que estão resolvendo alguma questão e colam o enunciado e idéia da solução no corpo do código.

E aí, vai escrever 20 vezes as duas barras // ?
Claro que não, use os delimitadores '/*' e '*/'.

Tudo o que você quer ver comentando coloca entre /* e */, e tudo que será dentro será ignorado.

A diferença desses delimitadores para as duas barras //, é que as barras ignoram o que tem naquela linha, já os delimitadores /**/ ignoram TUDO que estejam entre eles, seja uma letra, linha ou milhões de linhas.

Veja um exemplo que possui dezenas de linhas entre os delimitadores /**/, mas ao compilarmos e executarmos esse código em C, somente o *printf* é que faz realmente algo:

```
#include <stdio.h>

int main()
{
/* Adoro estudar a linguagem de programação C enquanto escuto metal!
Iron Maiden - Fear Of The Dark
I am a man who walks alone
And when I'm walking in a dark road
At night or strolling through the park

When the light begins to change
I sometimes feel a little strange
A little anxious when it's dark

Fear of the dark, fear of the dark
I have a constant fear that something's always near
Fear of the dark, fear of the dark
I have a phobia that someone's always there

Have you run your fingers down the wall
And have you felt your neck skin crawl
When you're searching for the light?
Sometimes when you're scared to take a look
At the corner of the room
You've sensed that something's watching you

Fear of the dark, fear of the dark
I have a constant fear that something's always near
Fear of the dark, fear of the dark
I have a phobia that someone's always there

Have you ever been alone at night
Thought you heard footsteps behind
And turned around and no one's there?
```

```
And as you quicken up your pace
You'll find it hard to look again
Because you're sure that someone's there

Fear of the dark, fear of the dark
I have a constant fear that something's always near
Fear of the dark, fear of the dark
I have a phobia that someone's always there

Watching horror films the night before
Debating witches and folklores
The unknown troubles on your mind
Maybe your mind is playing tricks
You sense, and suddenly eyes fix
On dancing shadows from behind

(2x)
Fear of the dark, fear of the dark
I have constant fear that something's always near
Fear of the dark, fear of the dark
I have a phobia that someone's always there

When I'm walking in a dark road
I am a man who walks alone */
    printf("Colocamos a letra da música Fear of the Dark, da banda Iron
Maiden - mas voce nao viu !)\n");
}
```

- **Use comentários com moderação**

Embora seus comentários não prejudiquem o funcionamento de seu programa em C, evite usar ele desnecessariamente, comentando *printf* e outras coisas óbvias: 'agora exibimos a mensagem', 'agora somamos os números' etc.

Use comentários para explicar o que é o programa, para que serve, qual a utilidade de um algoritmo que é mais complexo, elaborado ou grande.

O tipo de dado inteiro (int) na Linguagem C

Vamos estudar agora um dos tópicos mais básicos, bastante usado e importante na linguagem de programação C: os tipos de dados.

Para começar, vamos iniciar falando sobre os números inteiros em C, um tipo de dado muito importante, que usaremos durante toda nossa **apostila de C**.

Como declarar variáveis inteiras na linguagem C

Variáveis são os nomes que vamos dar a determinado bloco de memória. Sempre que você quiser usar um dado (um número, um caractere, um texto etc) na linguagem C, você precisará declarar uma variável.

Ao fazer isso, você estará selecionando um espaço na memória (lembre-se que o C é uma linguagem poderosíssima e age no hardware de sua máquina).

Por exemplo, suponha que você queira armazenar sua pontuação em um jogo que você desenvolveu em C.
Se esse valor for inteiro, você precisará declarar uma variável (poderia usar o nome 'pontos', por exemplo), e o C irá alocar um espaço de 2 ou 4 bytes de memória. Aquele espaço em memória serve somente para sua variável, é o endereço dela.

Para declarar uma variável inteira fazemos:

int nome_de_sua_variavel;

Ao fazer isso, estamos reservando um espaço em memória para guardar um número. Em vez de lidarmos com o número da posição da memória, vamos usar seu nome, que é nome_de_sua_variavel;

Vale realçar que os tipos (no caso, usamos o tipo *int*), são sempre em letras minúsculas e vêm antes do nome da variável.

- **Como imprimir um número inteiro na tela, usando a linguagem C**

Vamos mostrar agora como imprimir o valor de uma variável na função printf. Dentro do printf, entre as aspas duplas, os números são representados por: %d, e após o fechamento das aspas, colocamos uma vírgula e o real valor da variável que é representada por %d.

Por exemplo, vamos supor que você queira fazer um programa que imprima na tela: "Eu tenho 20 anos", mas sem digitar o 20.
Ele ficaria assim:

```
#include <stdio.h>

int main()
{
   printf("Eu tenho %d anos", 18);
}
```

Podemos colocar vários números inteiros representados por %d dentro das aspas, e seus valores serão os que vem após a vírgula, na mesma ordem. Por exemplo, um programa em C que imprima "Eu nasci em 2112, no dia 21 do mes 12":

```
#include <stdio.h>

int main()
{
   printf("Eu nasci no ano %d, no dia %d do mes %d", 2112, 21, 12);
}
```

Agora que já sabemos declarar um número e imprimir ele na tela, vamos fazer os dois.
Vamos criar uma variável inteira chamada 'cprogressivo' e imprimir seu valor no printf:

```
#include <stdio.h>

int main()
{
   int cprogressivo;
```

```
    printf("O valor da variavel 'cprogressivo' eh %d", cprogressivo);
}
```

Aqui notamos duas coisas interessantes:
1. Digitamos 'cprogressivo' dentro do escopo da função printf, e o que aconteceu? Apareceu o nome 'cprogressivo', e não o valor dessa variável inteira. Como dissemos, para aparecer o valor dentro de um printf, usamos o %d.

2. Apareceu um número totalmente aleatório e louco (conhecido como lixo, em C), não foi? Por quê?

Ao declarar a variável 'cprogressivo', nós reservamos um espaço em memória. Dentro da printf nós imprimos o seu conteúdo, e naquela posição da memória de seu computador havia aquele número lá armazenado (não me pergunte o porquê, mas saiba que pro computador é tudo número: textos, imagens, som etc).
No meu printf apareceu o número 2130567168, e na sua máquina?

- ## Como inicializar variáveis inteiras na linguagem C

Não nos serve para nada aqueles números aleatórios que você imprimiu com o último código.
Variáveis só servem se pudermos controlar seu valor, e para isso temos que inicializá-las, ou seja, colocar um valor dentro dessa variável.
Para isso, usamos o símbolo de igualdade '='.

A inicialização pode ocorrer de duas formas:
1. Junto com a declaração da variável

```
int idade = 18;
```

2. Depois da declaração, em qualquer lugar do programa (após a variável ter sido declarada)

```
int idade;
...
idade = 18;
```

Por exemplo, um programa que mostre na tela o salário que você desejar receber:

```c
#include <stdio.h>

int main()
{
    int salario = 5;
    printf("Eu pretendo ganhar %d mil reais como programador C", salario);
}
```

Note que você pode alterar o valor da variável 'salario', e esse valor sempre será impresso no printf. Qual o salário que você colocou?

Em breve vamos estudar em detalhes as operações matemáticas na Linguagem C, mas por hora vamos mostrar como é possível usar alguns cálculos com e nas variáveis inteiras, dentro ou não do printf.
Leia o seguinte programa e tente entender cada trecho do seguinte código, que está comentado.
Ele mostra que é possível declarar variáveis inteiras em várias partes do código, que podemos fazer operações matemáticas com as variáveis, dentre outras coisas.

```c
#include <stdio.h>

int main()
{
    int soma = 1+1;
    printf("1 + 1 = %d \n", soma);

    int numero1 = 10;
    int numero2 = 20;

    soma = numero1 + numero2; //Podemos fazer o valor de uma variável mudar no decorrer
                              //de um programa
    printf("%d + %d = %d \n", numero1, numero2, soma);

    //Ao invés de criar uma variável para armazenar um resultado de operação matemática,
    //podemos colocar direto esses cálculos dentro do printf
    printf("%d - %d = %d\n", soma, numero1, (soma-numero1));
}
```

Modificadores do tipo inteiro - short, long, signed e unsigned

Trabalhar com a linguagem de programação C é gratificante devido ao poder e liberdade que você tem, já que podemos atuar no 'talo' do sistema, inclusive selecionando espaços de memória.

Porém, isso tem um custo: estudo e cuidados adicionais, em relação a grande maioria das outras linguagens de programação.

Por exemplo, o tamanho que um variável do tipo inteira (int) pode ocupar em diferentes computadores e sistemas. Falaremos mais detalhes sobre essas variações neste artigo de nossa **apostila de C**.

- **O tamanho que cada variável em C pode ocupar em memória**

Vimos que uma variável do tipo int (inteira) em C, geralmente ocupa 2 ou 4 bytes na memória de seu computador.

Essa diferença, à priori, pode parecer insignificante. Mas é porque estamos no início de nossos estudos na linguagem C, e por hora, nossos códigos e programas são bem simples e pequenos.

Mas a maior parte dos sistemas operacionais, como o Linux e o Windows, são feitos em C e aí essa diferença pode se tornar um problema, principalmente por questões de portabilidade.

Ou seja, você pode compilar/rodar seu código em C no seu computador e obter um resultado, mas pode rodar em outro computador ou sistema (de diferentes configurações) e obter resultados totalmente diferentes (e erros, as vezes).

O grande diferencial da linguagem de programação Java é que ela não roda em sua máquina, e sim em uma máquina virtual.
Essa máquina virtual (JVM - Java Virtual Machine) é simulada em todos os dispositivos, portanto, o Java roda da mesma maneira em todos os sistemas (porém, geralmente suas aplicações são bem mais lentas que aquelas feitas em C ou C++, por exemplo).

Para saber mais sobre Java, oferecemos um curso completo de Java no site Java Progressivo: http://www.javaprogressivo.net/

Se quiser saber quanto vale o valor do int, ou de qualquer outra variável, use a função 'sizeof(tipo)', e troque 'tipo' por um tipo de variável.

O exemplo a seguir mostra como descobrir o valor da variável int em C, usando a função sizeof:

```
#include <stdio.h>

int main()
{
   printf("O tamanho do inteiro em meu computador: %d bytes", sizeof(int));
}
```

- **Tendo um maior controle sobre o tamanho dos inteiros - short e long**

Vamos apresentar agora dois modificadores do tipo inteiro em C: short e long, ele alteram o tamanho de bytes do inteiro.
A diferença entre os inteiros e modificadores está na faixa de valores armazenadas.

Um inteiro de 1 byte (char) armazena do número -128 a +127
Um inteiro de 2 bytes armazena do número -32 768 a +32 767
Um inteiro de 4 bytes armazena do número -2 147 483 648 a +2 147 483 647
Um inteiro de 8 bytes armazena do número -9 223 372 036 854 775 808 a +9 223 372 036 854 775 807

'short' em inglês, significa curto e 'long', longo.
Colocando uma dessas palavras antes da 'int', você definirá um tamanho e uma faixa de valores para suas variáveis.

Por exemplo, se criar a variável inteira 'numero' como short, deverá fazer:

```
short int numero;
```

De modo análogo para uma variável com o modificador 'long':

```
long int numero;
```

Para saber o tamanho de bytes de seu sistema, use novamente a função sizeof:

```
#include <stdio.h>

int main()
{
    printf("int : %d bytes\n", sizeof( int ) );
    printf("short int: %d bytes\n", sizeof( short ) );
    printf("long int: %d bytes\n", sizeof( long ) );
}
```

- **Controlando a faixa de valores dos inteiros através do sinal: signed e unsigned**

Todos nós sabemos que os números, como os inteiros, podem assumir tanto valores negativos como positivos.
Porém, muitas vezes, valores negativos (ou positivos) podem ser inúteis, chegando até a atrapalhar em termos de computação.

Por exemplo, o tamanho de memória é sempre positivo, não existe um bloco de -2 bytes em sua máquina.
Então, para declarar que um número seja apenas positivo (incluindo o 0), usamos o modificador unsigned:
unsigned int teste;

Analogamente para especificar que o inteiro possui valores positivos e negativos:
signed int teste;

Em C, por padrão, ao declararmos uma variável do tipo int, ele será automaticamente do tipo signed.
Portanto, a declaração passada é inútil, serve apenas para fins didáticos.

Caso queiramos apenas números positivos, a faixa de valores negativos que apresentamos é inútil e podemos desconsiderar ela, aumentando a faixa de valores positivos.

Então, ao declararmos inteiros com o modificador unsigned, as faixas de valores passam a ser:
Um inteiro de 1 byte (char) armazena do número 0 a +255
Um inteiro de 2 bytes armazena do número 0 a +65 535
Um inteiro de 4 bytes armazena do número 0 a +4 294 967 295
Um inteiro de 8 bytes armazena do número 0 a +18 446 744 073 709 551 615

- **Quando usar short, long, signed e unsigned**

O curso C Progressivo visa ensinar o básico, então, não será necessário usarmos esses modificadores ao longo de nossa apostila online.

No futuro, quando você for um programador profissional, será um ótimo e diferenciado costume usar esses modificadores.

- **Quando usar o short int em C**

Você deverá usar o short quando quiser armazenar, sempre, valores pequenos.

Por exemplo, suponha que você vá fazer um banco de dados para o Governo Federal, onde terá que criar milhões de cadastros.
Para armazenar idades, usar short int. Ora, valores inteiros de idades são pequenos e raramente passam do 100.
Então não desperdice memória à toa, use short!

- **Quando usar o long int em C**

O long é bastante usado para cálculos de cunho acadêmico, como científico e estatístico.

É comum também usarmos o long int para armazenar números como RG e CPF, que são compostos de vários dígitos.

- **Quando usar unsigned int em C**

Você pode usar o unsigned para se certificar que essa variável inteira nunca irá receber um valor negativo, como para armazenar dados de memória, idade, o life de um jogo etc.

Para saber mais sobre esse limites e padrões da linguagem C, acesse: http://en.wikipedia.org/wiki/Limits.h

- **Exercício:**

Crie um programa em C que mostre o tamanho das seguintes variáveis em seu computador:

- int
- short int
- long int
- signed int
- unsigned int
- short signed int
- short unsigned int
- long signed int
- long unsigned int

Os tipos float e double - números decimais (ou reais) em C

No artigo passado de nosso curso de C, estudamos sobre o tipo inteiro (int), como declarar, imprimir e inicializar tal tipo de dado.

Agora faremos o mesmo, mas para os números decimais, também conhecidos como números reais, que são os tipos float e double.

- **O que são e para que servem os tipos float e double**

Em nosso dia-a-dia, muitas vezes usamos dados em forma de números decimais, como por exemplo:
o rendimento da poupança (0,57%, 1,01%),
valores monetários (R$ 1,99) ,
as notas da faculdade (10,0 , 5,4 , 0,5),
constantes e outrs números matemáticos (pi = 3,14...) etc.

Não podemos, porém, armazenar essas informações em variáveis inteiras na linguagem C.
Ao invés disso, precisamos declarar usando os tipos de dados float e double, que são tipos especialmente feitos para que possamos trabalhar com números reais (decimais).

Vamos aprender como usar tais tipos de variáveis neste artigo de nossa apostila de C.

- **Como declarar e inicializar variáveis do tipo float e double na linguagem C**

Lembre-se que, para inteiros, declaramos da seguinte maneira: int idade, int mes etc.

Analogamente, para floats e double:

```
float pi;
float juros;
double tamanho_de_uma_bacteria;
double area_de_uma_circunferencia;
```

A declaração também não é diferente da que fizemos com inteiros.
Podemos inicializar valores tanto na declaração das variáveis como somente depois:

```
float pi = 3.14;
double juros = 1.32101;
```

Ou

```
float pi;
double juros;
pi = 3.14;
juros = 1.32101;
```

Se você é novo no mundo da programação e nunca teve contato com uma língua estrangeira, certamente achará tais inicializações de variáveis estranha. Mas esse é uma regra bem importante:
Na linguagem de programação C, usamos o PONTO (.), e não a vírgula para separar a parte inteira da decimal.

Ou seja, no Brasil escrevemos: 1,99 e 0,57
Em programação é: 1.99 e 0.57

- ## Qual a diferença entre float e double

Em nossos exemplos, usamos tanto float como double para representar números reais.
Temos, então, duas opções iguais para representar esses números decimais? Na verdade não, há uma diferença.

Variáveis float exigem, geralmente, 4 bytes de memória para serem armazenadas enquanto double necessitam de 8 bytes.
Essa diferença serve para termos uma melhor precisão na hora de realizar cálculos.

O número PI, por exemplo, é irracional. Ou seja, ela possui uma quantidade INFINITA de casas decimais.
Obviamente, uma cálculo com o uso do pi nunca é totalmente preciso. Além do mais, computadores tem uma quantidade de memória limitada.

Então, nos seus trabalhos escolares você deve declarar e usar uma variável do tipo float para representar o número pi:
float pi = 3.14;
Se quiser ser mais preciso pode fazer até: pi = 3.1415;

Já um Engenheiro Civil ou um Físico da NASA terá que usar uma precisão maior, pois quanto mais casas decimais, mais correto

será seu resultado. Então, eles usariam:
double pi = 3,14159265358979323

Ok, agora você sabe a diferença entre um float e um double - apenas a precisão.
Mas qual a diferença entre 0 e 0.0? E a diferença de 1 e 1.00?

Você sabe que um inteiro ocupa 2 bytes na memória, e que um float ocupa 4 bytes.
Além do tamanho alocado em sua máquina, qual outra diferença que faz esses valores diferentes?
Sim, o ponto. Ou seja, a parte decimal.

Fazer: int erro = 0
É totalmente diferente de: float erro = 0.0;

Uma vez declarado um inteiro, não poderá usar decimais nele. Mesmo sabendo que 0 = 0.0
Já os decimais podem ser trabalhados com inteiros.
Por exemplo:
double erro = 0.00
int juros = 1

Podemos fazer: juros + erro = 1.00
Ou seja, quando fazemos uma operação matemática de um decimal com inteiro, obteremos sempre um decimal.

Assim, o resultado dessa operação deverá sempre ser armazenado em um float ou em um double.

Veremos mais sobre isso quando estudarmos operações matemáticas na linguagem C.

- **Imprimindo números reais float e double na tela através do printf**

Vimos na aula passada que representamos inteiros como %d dentro das aspas, de um printf.

Para variáveis decimais ou reais, como o float e o double usamos: %f
Vejamos um exemplo que mostra um valor de pi com precisão simples (float) e outro com precisão dupla(double):

```
#include <stdio.h>

int main()
{
    float pi = 3.14;
    double piDouble = 3.14159265358979323846264338327795;
    printf("Valor de pi %f\n", pi );
    printf("Valor de pi mais preciso %f\n", piDouble );
}
```

Aqui notamos uma coisa curiosa no segundo valor, é exibido: 3.141593
Ou seja, o C não mostrou todo o valor da variável double 'piDouble' e ainda arredondou!

Podemos resolver isso da seguinte maneira. Supondo que você queira que seja exibido 6 casas decimais:
Ao invés de usar '%f' coloque: '%.7f'
Ou seja, esse 0.7f diz ao C o seguinte "Após o ponto, exiba 7 casas decimais".
Teste e veja o resultado:

```
#include <stdio.h>

int main()
{
    float pi = 3.14;
    double piDouble = 3.14159265358979323846264338327795;
    printf("Valor de pi %f\n", pi );
    printf("Valor de pi mais preciso %.7f\n", piDouble );
```

```
}
```

Será exibido: 3.1415927

Agora veja o seguinte: a variável 'pi' tem somente duas casas decimais depois do ponto.
O que ocorre se eu ordenar ao printf que imprima com 5 casas decimais?
Programe e veja você o que acontece:

```
#include <stdio.h>

int main()
{
    float pi = 3.14;
    double piDouble = 3.14159265358979323846264338327795;
    printf("Valor de pi %.5f\n", pi );
    printf("Valor de pi mais preciso %.7f\n", piDouble );
}
```

- **Como imprimir números na forma exponencial em C**

Outra maneira de imprimir variáveis decimais é usando exponenciais.

Podemos inicializar uma variável da seguinte maneira:
float numero = xEy;

Isso significa: x vezes 10 elevado a y = x * 10^y
Ou seja, 1E6 = 1 vezes 10^6 = 1 milhão

E float numero = xE-y
Significa: x vezes 10 elevado a -y = x * 10^(-y)
Por exemplo: 2E-3 = 2 vezes 10^(-3) = 0.002

Veja o seguinte código e tente adivinhar sua saída. Logo após, rode o programa para vê se acertou:

```c
#include <stdio.h>
int main()
{
    float salarioSonho = 1E6,
          salarioReal = 10E-3;
    printf("Sonhei que meu salario era de R$%.2f, \nmas acordei e lembrei que era %.2f centavos", salarioSonho, salarioReal);
}
```

Nesse último exemplo, note como declaramos mais de uma varíável.
Em vez de fazer:

```c
float variavel1;
float variavel2;
float variavel3.
```

Você pode fazer:

```c
float variavel1, variavel2, variavel3;
```

Ou, para ficar mais legível:

```c
float variavel1,
      variavel2,
      variavel3.
```

O tipo char

Agora que você já sabe como lidar com inteiros e decimais na linguagem C, está na hora de estudarmos como escrever caracteres.

- **Como declarar o tipo char em C**

Para armazenar caracteres vamos usar um tipo especial de dados, o char (de *character* - caractere, em inglês).

O tipo char serve para armazenar UM, e somente UM, caractere.
Para declarar, usamos a seguinte sintaxe;

```
char nomeDaVariavel;
```

Ao fazermos isso, estamos alocando 1 byte de memória para guardar nosso caractere.
Se você quiser armazenar mais caracteres, temos que usar as Strings, que são um conjunto de caracteres, usados para escrever textos maiores.

Nós estudaremos as Strings mais à frente, em nossa apostila de C.

- **Como inicializar variáveis char em C**

Para guardar uma letra no seu char, temos que fazer uma operação especial: sempre colocar o caractere entre aspas simples.

Por exemplo, para guardar a letra C, escrevemos:

```
char letra = 'C';
```

A sintaxe linguagem de programação é case sensitive, ou seja, minúsculo é diferente de maiúsculo.
Por exemplo:

```
char letraMinuscula = 'c';
char letraMaiuscula = 'C';
```

Esses dois caracteres, embora representem a mesma letra, são totalmente diferentes, pois uma é maiúscula e a outra é minúscula.

- **Como imprimir caracteres e textos na tela**

Vamos agora aprender como mostrar na tela letras e caracteres, através da função printf.

Assim como usamos %d para mostrar inteiros, %f para float e double, vamos usar um símbolo especial para caracteres: %c

Por exemplo, um programa que exiba o texto "C Progressivo" na tela tem o seguinte código:

```
#include <stdio.h>

int main()
{
    char letra0='C', letra1=' ', letra2='P', letra3='r', letra4='o', letra5='g',
    letra6='r', letra7='e', letra8='s', letra9='i', letra10='v',letra11='o', letra12='\n';
    printf("%c%c%c%c%c%c%c%c%c%c%c%c%c
%c",letra0,letra1,letra2,letra3,letra4,
    letra5,letra6,letra7,letra8,letra8,letra9,letra10,letra11,letra12);
}
```

Note 3 coisas interessante que mostramos:
1 - letra1=' '
Sim, espaço também é caractere!

2 - Repetimos letra8 duas vezes no printf, por quê?
Ora, a letra 's' se repete duas vezes na frase: "C Progressivo".
Em vez de gastar memória à toa para representar o mesmo caractere, o certo é repetir duas vezes o char.

3 - letra12='\n'
Sim, a quebra de linha, ou ENTER, é um caractere também!

Notou também a trabalheira e o tanto de código, apenas pra escrever isso? Inviável!
Em breve, em nosso curso C Progressivo, iremos aprender sobre strings, ou vetores de caracteres, que é uma maneira bem mais simples de se escrever e manipular textos em C.

- **A tabela ASCII em C**

Digite e rode o seguinte programa em C:

```
#include <stdio.h>

int main()
{
    char ascii = 67;
    printf("%c%",ascii);
}
```

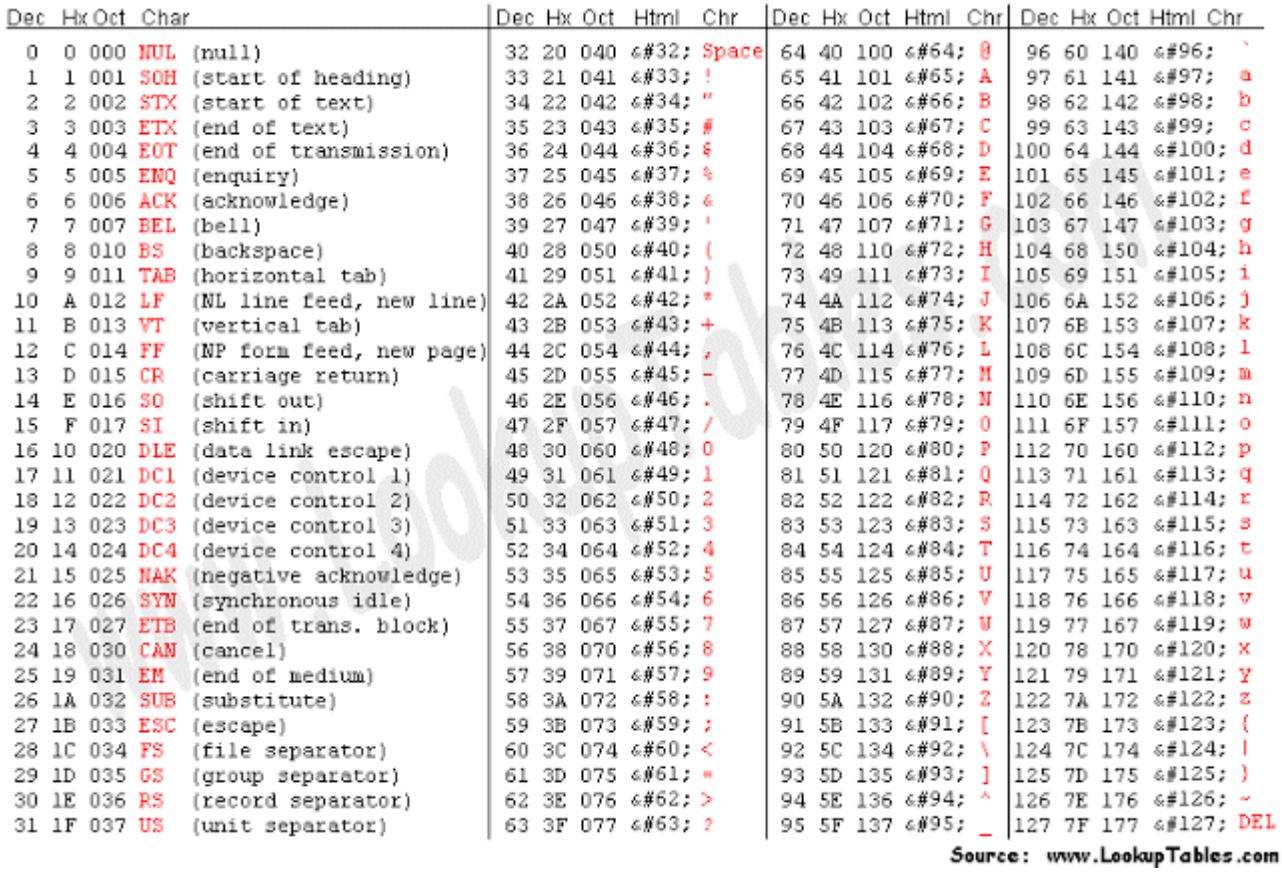
Estranho o resultado, não? Inicializamos um caractere sem aspas simples e digitamos dois números?
Se notar bem, não inicializamos o char com um caractere, pois não usamos aspas simples, iniciamos como se char fosse um inteiro.
Por que isso?

Ver char como inteiros não seria errado.
O que acontece é que cada caractere em programação recebe um número, como se fosse uma identificação.
A letra 'C' por exemplo, tem o número 67 como sua identificação.

Ou seja, para o computador:
char ascii = 67;
ou char ascii = 0x43;
ou char ascii = 'C';

É a mesma coisa. No primeiro caso, representamos o caractere como um número decimal.
No segundo, o caractere é representado pro um número no formato hexadecimal.
Altere o número para outro e veja os resultados.
Você verá que cada caractere, incluindo o ENTER, TAB ou aviso sonoro é identificado com um número.

A tabela ASCII nada mais é que uma lista completa dos números que identificam os caracteres, veja só:



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Podemos fazer o contrário também. Dentro do printc colocar os %c, e números após a vírgula.
Por exemplo, a letra 'o' pode ser representada pelo número 111 e a letra 'i' pode ser representada po 105.

Então podemos escrever 'oi' da seguinte maneira:

```
#include <stdio.h>

int main()
{
    printf("%c%c ",111,105);
}
```

Sim: 111,105 quer dizer 'oi'. Legal, não?
Agora você já pode escrever cartas e códigos secreto, e só quem sabe a poderosa linguagem C que pode ler.

Por exemplo, descubra o que está dito no programa abaixo:

```c
#include <stdio.h>

int main()
{
    printf("%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c
%c%c%c%c%c%c%c%c%c%c%c%c%c",

80,97,114,97,98,101,110,115,44,32,118,111,99,101,32,100,101,115,99,111,98
,114,105,117,32,111,32,115,101,103,114,101,100,111);

}
```

Exercício:

Escreva seu nome usando apenas a função printf, %c e números. Use a tabela ASCII.

A função scanf - recebendo números do usuário

Até o momento, os artigos de nosso curso C Progressivo tem mostrado diversos programas, porém todos estáticos, sem controle e sem interação.

O programa simplesmente roda e mostra coisas na tela.

Porém, na vida real não é assim. Praticamente todos possuem algum tipo de interação com o usuário: recebem dados, cliques, arrastamos e soltamos etc.

Nessa lição iremos aprender a usar a função scanf e obter dados do usuário. Vamos começar a 'conversar' com o computador.

Recebendo números inteiros com a função scanf

Assim como fizemos para trabalhar com inteiros na função printf, vamos usar novamente o símbolo %d para representar os int.

Suponho que queiramos pedir um inteiro ao usuário, primeiro temos que declarar um inteiro. Vamos supor um de nome 'numero'.
Para o usuário armazenar um número nessa variável 'numero', usamos a seguinte sintaxe:

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

Essa função nos diz "armazene na variável 'numero' um inteiro".
O erro mais comum é esquecer o &, cuidado!

Por exemplo, vamos escrever um programa em C que pede um número ao usuário e o mostra na tela:

```
#include <stdio.h>

int main()
{
    int numero;
    printf("Digite um numero: ");
    scanf("%d", &numero);

    printf("O numero digitado foi: %d", numero);
}
```

- **Exemplo de código- Como usar a função scanf()**

Escreva um programa em C que peça dois números inteiros e mostre sua soma.
Primeiro criamos três variáveis inteiras: num1, num2 e resultado.
Essas variáveis que vão armazenar os números que o usuário fornecer e o resultado da soma.

Após isso, usamos a função scanf() para pegar do usuário os dois número.
Em seguida, armazenamos a soma desses números na variável 'resultado', e exibimos essa variável num printf.

Veja como ficou o nosso código em C:

```c
#include <stdio.h>

int main()
{
    int num1, num2, resultado;
    printf("Digite um numero: ");
    scanf("%d", &num1);

    printf("Digite outro numero: ");
    scanf("%d", &num2);

    resultado = num1 + num2;

    printf("%d + %d = %d", num1,num2,resultado);
}
```

- **Recebendo mais de um número dentro de uma única scanf**

Para evitar o trabalho de ter que escrever a scanf cada vez que você deseje receber um número do usuário, você pode colocar mais de um %d dentro do escopo da scanf.

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

```
    int num1, num2;
    printf("Insira dois numeros: ");
    scanf("%d %d", &num1, &num2);
    printf("Você digitou: '%d' e '%d'", num1, num2);
}
```

No caso acima, num1 vai ser o número que você digitou antes de dar enter, espaço ou tab.
E num2 sera o número que você digitou após dar enter, espaço ou tab

Exercício: Faça um programa que peça dois inteiros ao usuário e que mostre a diferença (subtração) entre o primeiro e segundo número.

- **Recebendo números reais ou decimais com a função scanf**

Para receber números do tipo float ou double, fazemos exatamente como na seção anterior, sobre inteiros, somente com uma diferença que talvez você já saiba qual é: usamos %f ao invés de %d.
Faz todo sentido, não?

Exemplo: Crie um programa em C que peça dois números decimais ao usuário e mostre o produto deles, com precisão de dois números.

Lembrando que o símbolo de multiplicação é o asterisco, *:

```
#include <stdio.h>
int main()
{
    float num1, num2, resultado;
    printf("Digite um numero: ");
    scanf("%f", &num1);

    printf("Digite outro numero: ");
    scanf("%f", &num2);

    resultado = num1 * num2;

    printf("%.1f + %.1f = %.2f", num1,num2,resultado);
}
```

Exercício: Faça um programa em C que peça dois números do tipo double ao usuário e mostre o resultado da divisão do primeiro pelo segundo e exiba esse resultado com 3 casas decimais.
PS: o símbolo de divisão é /

Recebendo letras do usuário - As funções scanf, getchar, fgetc e getc

No artigo passado de nosso curso de C, aprendemos como usar a função scanf para receber dados do usuário, e com isso fazer programas bem mais dinâmicos.

Porém, lá tratamos só de números (tanto inteiros, como decimais). Mas, muitas vezes, nós precisamos digitar textos em programas.
Por isso vamos aprender como receber letras do usuário.

Vamos explicar o que são e como funcionam as funções scanf, getchar e fflush.

- **Recebendo caracteres em C através da função scanf()**

A maneira é idêntica a que fizemos quando usamos a função scanf() para capturar números digitados pelo usuário, que como ensinado no artigo passado de nossa apostila.

Mas com os caracteres, ao invés de usar %d (para números inteiros0 ou %f (para números floats), vamos usar %c.
Pois é assim que representamos variáveis do tipo char em C.

Portanto, um programa que pede um caractere ao usuário e o imprime na tela é feito da seguinte maneira:

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

    printf("Insira um caractere: ");
    scanf("%c",&letra);
    printf("Você digitou: '%c'", letra);
}
```

- **Recebendo caracteres em C através da função getchar()**

A função scanf() é bem poderosa e flexível.

Com ela, podemos pegar uma infinidade de dados do usuário, e inclusive escolher o que vai ser 'capturado', limitar o tanto de coisas que pode ser escrito, e uma série de outras funcionalidades que vocês irão aprender durante nosso curso de C.

Porém, há mais opções que facilitam nossa vida.

Existe uma função que faz o mesmo papel da scanf e é voltada para o uso com caracteres, é a getchar().
Ela é mais simples, pois não precisar usar %c ou &, como fazemos na scanf(), e foi feito especialmente para ser usado com caracteres.
get -> pegar
char -> caractere

Para usar, fazemos:
seu_caractere = getchar();

Veja como é seu uso em um código que pede um caractere para o usuário, armazena no char 'letra', e em seguida exibe esse mesmo caractere.

```
#include <stdio.h>

int main()
{
    char letra;

    printf("Insira um caractere: ");
    letra = getchar();
    printf("Você digitou: '%c'", letra);
}
```

- **Recebendo caracteres em C através das funçoes fgetc e getc**

Essas funções também servem para armazenar caracteres, porém são mais gerais que a getchar(), pois podem receber dados de outras fontes, além da padrão (o teclado - stdin).

Seus usos são semelhantes ao getchar(), porém temos que dizer ao C por onde o programa deve receber as informações.
No nosso caso, e pelo teclado, então fica: fgetc(stind) e getc(stdin)

Por exemplo, um aplicativo em C que recebe duas letras e mostra na tela seria:

```c
#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Insira um caractere: ");
    letra1 = getc(stdin);

    printf("Insira outro caractere: ");
    letra2 = getc(stdin);

    printf("Você digitou: '%c' e '%c'", letra1, letra2);
}
```

É importante que você faça esse exemplo e veja o resultado. Agora teste com as outras funções: scanf e getchar
Estranho esse problema, não?

Isso tem a ver com uma região especial de memória, uma espécie de memória temporária chamada buffer, que vamos estudar mais no próximo artigo do curso C Progressivo.

Buffer: o que é, como limpar e as funções fflush e __fpurge

No artigo passado foi pedido o seguinte programa:

Fazer um programa em C que peça dois caracteres ao usuário e os exiba.

Porém, há um problema ao se fazer isso, que e o que vai ser explicado nesse artigo de nossa **apostila de C**.

- **O problema de usar scanf, getchar, getc e fgetc para receber caracteres em C**

"Ora, é só declarar duas variáveis char e usar a scanf duas vezes, uma para cada variável", é que você deve ter pensado para criar o tal programa.

Vamos fazer isso então, e ver o resultado:

```
#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Insira um caractere: ");
    scanf("%c",&letra1);

    printf("Insira outro caractere: ");
    scanf("%c",&letra2);

    printf("Você digitou: '%c' e '%c'", letra1, letra2);
}
```

Eu digitei 'C' (a melhor letra do alfabeto), dei enter, e antes de digitar a próxima letra o progarma terminou, exibindo a seguinte mensagem:
"Você digitou: 'C' e '
'"

Nossa! Estranho, não? Será que hackeamos a linguagem C e descobrimos uma falha?
Não ;)

Note que digitei 'C' e enter...mas enter também é uma tecla, e é representada por '\n', lembra?
Ou seja, o C entendeu que nossa segunda letra era o enter!

A solução para isso é bem simples.
Na função scanf, dê um espaço entre a aspa " e o símbolo %c.
Nosso código fica assim:

```
#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Insira um caractere: ");
    scanf("%c",&letra1);

    printf("Insira outro caractere: ");
    scanf(" %c",&letra2);

    printf("Você digitou: '%c' e '%c'", letra1, letra2);
}
```

Pronto! Agora funciona perfeitamente!
Pois esse simples espaço é um comando para o C desconsiderar o enter, tab ou espaço em branco.
Sim, precisa nem dar enter. Tente aí digitar uma letra, dar espaço (ou tab) e a outra.

Ok, mas só é possível fazer isso na scanf().
E na getchar(), getc() e fgetc()?

- **Limpando o buffer em C: fflush e __fpurge**

Ainda no primeiro exemplo desse artigo (o que dá problema), digitamos a letra 'C', que é armazenada na variável 'letra1' e em seguida apertamos enter.

Esse caractere (enter), ficará armazenado no buffer do teclado (um memória temporária).

Em seguida, noss programa em C pede para que algo seja armazenado na variável 'letra2'.
Porém, antes do C receber um novo dado do usuário, ele checa se não tem mais alguma coisa armazenada no teclado (ele sempre faz isso...fez antes, para pegar a letra 'C'). E lá tem um caractere sim, o enter.

Então o programa pega esse caractere e o coloca na variável letra2, e é por isso que aparece uma quebra de linha em nosso programa.

Portanto, uma alternativa, caso não queria usar o espaço entre " e o %c na scanf, é **limpar o buffer** após cada scanf(), getchar(), getc() ou fgetc().

Para limpar o buffer em Windows, use: **fflush(stdin)**
Para limpar o buffer em Linux, use: **__fpurge(stdin)**

Veja como fica nosso programa original, funcionando do jeito que queríamos:

```
#include <stdio.h>
int main()
{
    char letra1, letra2;

    printf("Insira um caractere: ");
    scanf("%c", &letra1);

    fflush(stdin);
    __fpurge(stdin);

    printf("Insira outro caractere: ");
    scanf("%c", &letra2);

    printf("Você digitou: '%c' e '%c'", letra1, letra2);
}
```

Fica ao seu dispor escolher como vai ser.

E se habitue com essas coisas, em C há várias maneiras de se fazer várias coisas. Muitas vezes, porém, a solução que usamos não é muito segura ou portável, mas é a mais simples.

Limpar o buffer, por exemplo, nem sempre é algo desejável, e para programação mais profissional e segura não é recomendado que se use **fflush** por exemplo.

Mas para quem está começando, não há problema algum ficar limpando o buffer após cada scanf, e o scanf (embora seja arriscado e não indicado em alguns casos) é o mais usado.

Por isso, usaremos bastante o scanf ao longo de nosso curso de C.

Operações matemáticas em C - Soma, subtração, multiplicação, divisão e módulo (ou resto da divisão) e precedência dos operadores

Operações matemáticas básicas. Fácil não?

Por exemplo, quanto é: **1 + 1 x 2** ?

Pode ser 3: **1 + (1x2) = 1 + 2 = 3**
Ou pode ser 4: **(1+1)x2 = 2x2= 4**

Então, qual a resposta certa? Obviamente é só uma.
Para o computador, um cálculo não pode resultar em dois valores.

A resposta correta é sempre 3. Isso tem a ver com operações matemáticas e precedência em C.

- **Símbolos matemáticos em C**

Em programação C, alguns símbolos matemáticos são que usamos em computação são diferentes daqueles que usamos no dia-a-dia. Os símbolos são os seguintes:
Soma: +
Subtração: -
Multiplicação: *
Divisão: /
Módulo ou resto da divisão: %

Como pode notar, o símbolo de multiplicação é um asterisco, e não um 'x'.
O símbolo da divisão é a mesma barra que usamos para digitar urls.
Sobre módulo (ou resto da divisão), falaremos melhor sobre esse operador matemático ainda nesse artigo de nosso curso.

A seguir, um programa que pede dois números para o usuário e mostra o resultado da soma, subtração, multiplicação e divisão do primeiro pelo segundo número.
Note que é bem simples de se entender e não há segredo:

```c
#include <stdio.h>

int main()
{
    float num1, num2, sum, sub, mult, div;

    printf("Digite o primeiro numero: " );
    scanf("%f", &num1);

    printf("Digite o segundo numero: " );
    scanf("%f", &num2);

    //Soma
    sum = num1 + num2;

    //Subtração
    sub = num1 - num2;

    //Multiplicaçao
    mult = num1 * num2;

    //Divisão
    div = num1/num2;

    printf("%.2f + %.2f = %.2f\n", num1, num2, sum);
    printf("%.2f - %.2f = %.2f\n", num1, num2, sub);
    printf("%.2f * %.2f = %.2f\n", num1, num2, mult);
    printf("%.2f / %.2f = %.2f\n", num1, num2, div);
}
```

Além dos dois números que pedimos ao usuário, criamos mais 4 float para armazenar as operações matemáticas. Fizemos isso para o código ficar organizado, mas você também pode colocar essas operações direto no printf, veja como ficariam os printf:

```c
printf("%.2f + %.2f = %.2f\n", num1, num2, num1 + num2);
printf("%.2f - %.2f = %.2f\n", num1, num2, num1 - num2);
printf("%.2f * %.2f = %.2f\n", num1, num2, num1 * num2);
printf("%.2f / %.2f = %.2f\n", num1, num2, num1 / num2);
```

- **O que é módulo ou operador resto da divisão**

Vamos voltar um pouco no tempo, quando você aprendeu a fazer continhas, mais ou menos como na figura:

Pois é, nesse caso o resto da divisão é o 14.
Ela é resultado da seguinte operação: 2009 % 19 = 14
Pois 2009 = 19*105 + 14

Essa operação será bastante usada no decorrer de sua vida como programador C.
Por exemplo, os números pares sempre deixa resto 0 quando divididos por 2 (consegue entender o motivo?).

- **Precedência dos operadores matemáticos em C**

Vamos voltar ao exemplo do início de nosso tutorial: quanto seria 1 + 1 x 2
A resposta é sempre 3 pois o operador de multiplicação é mais importante que a soma.

E caso fosse: 8/4 - 2?
Pode ser: 8/4 - 2 = 2 - 2 = 0
Ou pode ser: 8/4 - 2 = 8/(4 - 2) = 8 / 2 = 4
A resposta é sempre 0, pois o operador de divisão é mais importante que o de subtração.

Mostramos esses exemplos para provar que saber a ordem (ou precedência) dos operadores é muito importante.
Embora a maioria dos cursos (alguns ditos 'bons' e famosos) simplesmente ignore isso, mas você viu que pode obter respostas totalmente diferentes.

Segue a lista dos operadores matemáticos que estudamos em nosso artigo. A importância vai crescendo de **baixo para cima** e da **esquerda para a direita:**

*** / %**
+ -

- **Usando parênteses para evitar confusão com a precedência**

Caso não seja muito fã ou bom em memorização, existe um recurso que você pode usar que vai deixar claro a ordem das operações, bem como deixar mais organizado seu código em C.
A regra é simples: agrupe e ponha entre parênteses o que você quer calcular.

Por exemplo, se ao invés de 1 + 1 x 2, tivéssemos escrito:
1 + (1 x 2) ?
É bem óbvio que somamos o 1 com o RESULTADO do que está entre parênteses. Ou seja, sempre é calculado primeiro o que está em parênteses:
1 + (1 x 2) = 1 + 2 = 3
(1 + 1) x 2 = 2 x 2 = 4

O outro caso 8/4 - 2, também fica bem evidente e simples de se entender apenas olhando:
(8/4) - 2 = 2 - 2 = 0
8/(4 - 2) = 8/2 = 4

E você achando que sabia as operações básicas de Matemática, hein?

Atalhos com símbolos matemáticos em C: += , -=, *=, /= e %=

Essa é uma das lições mais simples e rápida, tanto dessa seção sobre conhecimentos básicos da linguagem de programação C, como do curso C Progressivo inteiro.

Vamos aprender atalhos, maneiras mais rápidas e eficientes de escrever as operações matemáticas em C que você aprendeu no artigo passado de nossa **apostila online de C**.

Um dos artifícios mais usados pelos programadores C, que parece um pouco estranho na primeira vez que vemos, são umas abreviações usadas para descrever as operações matemáticas envolvendo uma mesma variável.

Vamos ver, em detalhes, cada uma dessas abreviações.

- **Fazendo contas com o valor antigo da variável**

Suponhamos que declaramos uma variável inteira de nome ***teste*** e inicializamos ela com o valor 0:

```
int teste = 0;
```

Agora vamos fazer com que essa variável tenha valor 1:

```
teste = 1;
```

Agora vamos fazer com que seu valor seja aumentado em 2:

```
teste = 3;
```

Agora vamos fazer com que seu valor seja aumentado em 2, novamente:

```
teste = 5;
```

Simples, não? Até aqui, nada demais.
Mas tem uma coisa que você não deve ter notado: foi necessário que você, programador, fizesse as contas.

"Mas peraí! Um computador é uma máquina que computa, ou seja, ela serve para contar e eu é que tenho que fazer os cálculos?"

Se quisermos adicionar 1 vao valor inicial da variável, é melhor fazermos:
teste = teste + 1;

Ou seja, a variável teste vai receber um novo valor. Que valor é esse?
É o valor 'teste + 1', onde esse 'teste' é o valor anterior.

Ou seja, antes teste=0,
Depois de: teste = teste + 1 , a variável será 'teste=1'.
É como se tivéssemos feito: teste = 0 + 1 = 1

Agora vamos adicionar 2 unidades ao valor de 'teste':
teste = teste + 2;

Como 'teste', anteriormente, era 1, agora 'teste' recebe valor 3:
teste = teste + 2 = 1 + 2 = 3;

E como fazemos para adicionar 3 unidades ao valor de 'teste'?
Fácil: teste = teste + 3;

Ou seja, estamos adicionando unidades, sem se importa e sem se estressar com o valor anterior da variável.

O mesmo é para subtração, multiplicação, divisão e resto inteiro da divisão.
Suponhamos que queiramos fazer uma operação com a variável 'x', podemos fazer:
x = x + 2;
x = x - 1;
x = x * 2;
x = x / 2;
x = x % 3;

Agora vamos para os atalhos!

- **Atalhos matemáticos: += , -= , *= , /= e %=**

+=

Em vez de escrever:
x = x + 2;

Podemos escrever:
x += 2;

-=

Em vez de escrever:
x = x - 1;

Podemos escrever:
x -= 1;

*=

Em vez de escrever:
x = x * 2;

Podemos escrever:
x *= 2;

/=

Em vez de escrever:
x = x / 2;

Podemos escrever:
x /= 2;

%=

Em vez de escrever:
x = x % 2;

Podemos escrever:
x %= 2;

Como podem ver, são notações e atalhos simples, mas extremamente úteis e usadas em nosso curso online e gratuito C Progressivo.

Sistema binário e Valores lógicos true ou false

Nessa simples aula de nosso curso C Progressivo, vamos falar sobre o sistema de numeração binária e dos valores lógicos, ou booleanos, true ou false.

Este é um artigo muito importante de nossa **apostila de C**, pois o sistema binário é de suma importância no ramo da programação.

Computadores e o sistema binário

Se já teve algum contato, por menor que seja, com computação, já deve ter ouvido falar no sistema de numeração binária.
Ele é a base de toda a computação, e, obviamente, da programação também. Ou seja, tudo que envolve linguagens de programação e hardware, tem a ver com o sistema binário.

E podemos ir mais além, tudo na tecnologia moderna, como seu computador inteiro, tablet, celular, TV Digital e o que mais existir, é baseado no sistema binário.

No dia-a-dia usamos o sistema decimal, podemos representar qualquer usando os algarismos: 0, 1, 2, 3, 4, 5, 6, 7, 8, e 9.
Já os computadores armazenam todo tipo de informação apenas por combinações dos números 1 ou 0.

Porém, não se assuste, não vamos entrar a fundo na matemática, só é preciso que saiba que para os computadores tudo se resume a números, tudo é bit: fotos, vídeos, textos, programas etc.

- **Valores lógicos: true ou false, 1 ou 0**

E onde o sistema binário entra na programação C?
Pois bem, diferentes de outras linguagens, como Java e C#, a linguagem de programação C é de baixo nível.

Isso quer dizer que vamos entrar mais a fundo nos nossos computadores. Com a linguagem C vamos ter acesso, por exemplo, aos espaços de memória de nosso sistema, por isso não se assuste quando falarmos em bits com certa freqüência.

Como dito anteriormente, tudo se resume a 1 ou 0.
Porém, vamos dar outro significado a isso: de agora em diante o 0 representa "falso" e tudo diferente de 0 será, obviamente, "verdadeiro" (true).

Por exemplo: 1 + 1 = 2?
Resposta: verdadeiro. Como representamos algo verdadeiro em valores lógicos?
Através de qualquer número diferente de 0: 1, 2, 3, 1 milhão, -10 etc.

E 1 + 1 = 3?
Como representamos isso em valor lógico? Apenas de uma maneira: 0

- **Exemplo prático dos valores lógicos em computação**

Com certeza você já deve ter usado algum programa que deu algum erro (principalmente se você usa o sistema operacional Windows).
Como bem deve se lembrar, deve ter visto algo do tipo: "Erro 144", "Error 221", "Problem: error 404", etc.

Pois bem, em programação quando um programa roda, ele geralmente termina dando um resultado, em forma de número.
O mais comum é que quando o programa termina sem mais problemas, ele retorne o número 0.

Caso o programa termine com algum erro, ele geralmente retorna outro valor diferente de 0.
Por exemplo, caso tenha faltado energia e o programa tenha fechado subitamente, ele resulta no número 1.
Caso o sistema trave, ele gera o número 2.
Caso você digite algo que não deveria, ele produz o número 3.

Pra que tudo isso? Comunicação entre programas.
Sabendo o número de retorno, outros programas ou o próprio sistema saberá exatamente o que ocorreu, pois cada número quer dizer que ocorreu algo diferente.

Usaremos muito, mas muito mesmo esses valores lógicos durante nosso curso de linguagem C.

Operadores Lógicos E (&&), OU (||) e de Negação (!)

Vimos no artigo passado de nossa apostila de C, uma explicação sobre valores lógicos true ou false em computação.

Vamos agora entrar mais em detalhes e ver como é possível representar informações apenas com true ou false, 1 ou 0.

E bem como o artigo passado, usaremos apenas ideias e conceitos, nada de código por hora.

- **Operador lógico 'E' em linguagem C: &&**

Vamos supor que você passou em um concurso, o cargo é para ser programador e trabalhar para o governo.
No edital você leu: é necessário ser brasileiro e ser maior de 18 anos.

Ou seja, você só pode fazer esse concurso se for brasileiro E TAMBÉM se tiver de 18 anos ou mais.
Se for brasileiro mas tiver menos de 18 anos, não pode.
Se tiver mais de 18 anos mas não for brasileiro, também não pode.

Ou seja: todas as condições devem ser obedecidas para termos um resultado positivo (que no caso é poder fazer o concurso).

Vamos agora trazer para o nosso mundo da programação em linguagem C. Esse E, vamos representar por &&

Então, para fazer o concurso: (ser brasileiro) && (ter 18 anos ou mais)
Vamos supor que você é brasileiro. Vamos representar esse fato como '1', pois é verdade.
Vamos supor que você tem 18 anos, representaremos isso pelo valor lógico '1' também.

Então: 1 && 1 resulta em verdade, ou 1.

E se você for menor?

Nossa expressão fica: 1 && 0, que resulta em resultado negativo, ou 0. Então você não pode fazer o concurso.

- **Operador lógico 'OU' em linguagem C: ||**

Vamos supor que você quer trabalhar em outro país, no Canadá por exemplo. Para isso ser possível você deve ser canadense OU ser casado com uma canadense.

Se você for canadense, mas não for casado com uma, pode trabalhar? Sim, pode. Se você não for canadense, mas for casado com uma canadense, pode trabalhar lá? Sim, pode.

Pode porque para isso ser possível (para ser verdade), é necessário ser canadense OU ser casado com uma.
Preenchendo apenas um dos requisitos, você está apto e o resultado é positivo.
E se você for canadense e casado com uma canadense? Ora, é óbvio que pode também.

Esse OU será representado em programação pelo símbolo: ||
Então: "1 || 0" resulta em valor lógico verdadeiro (true, ou 1)
"0 || 1" também, assim como "1 || 1" também vai resultar em valor lógico verdadeiro.

Assim, para expressões que possuem a condição OU (||), se uma das condições for verdadeira, toda a sentença será.
E para a expressão ser falsa, todas as condições devem ser falsas.

- **Operador lógico de negação em linguagem C: !**

Agora está bem claro pra você que as condições e expressões ou são verdadeiras (true ou 1) ou são falsas (false ou 0).

Quando queremos negar algo, colocamos o símbolo ! antes do que queremos negar.
Por exemplo, 1 + 1 é dois?

A resposta é sim, ou '1'. Ou '!0', pois o contrário de 0 é 1.
2 + 2 é três? Falso ou 0. Ou '!1', pois o contrário de 1 é 0
Assim como a lição passada, essa é bem simples, não?

Exercícios básicos sobre a linguagem C

As seguintes questões foram extraídas do material '300 ideias para Programar', de Virgílio Vasconcelos Vilela 10 questões pra você resolver, sobre saída.

Para que você aprenda bem a linguagem C e seja um programador profissional, é de suma importância que tente fazer todos os exercícios de nossa apostila de C.

Use a função printf para resolver os seguintes exercícios:

1. Frase na tela
gente nunca esquece!".

2. Etiqueta - Elabore um programa que pede seu nome, endereço, CEP e telefone. Ele deve imprirmir seu nome completo na primeira linha, seu endereço na segunda, e o CEP e telefone na terceira.

3. Frases assassinas - Faça um programa que mostre na tela algumas frases assassinas, que são
Eis alguns exemplos (bole também os seus):
"Isto não vai dar certo"
"Você nunca vai conseguir"
"Você vai se estrepar"
"Não vai dar em nada"
"Está tudo errado!"

4. Mensagem - Escreva uma mensagem para uma pessoa de que goste. Implemente um programa

5. Ao mestre - Escreva um bilhete ao seu professor, informando seus objetivos nesta disciplina e o
que espera dela e do professor. Implemente um programa que mostra seu bilhete na tela.

6. Quadrado - Escrever um programa que mostre a seguinte figura no alto da tela:

```
XXXXX
X   X
X   X
X   X
XXXXX
```

7. Tabela de notas - Escreva um programa que produza a seguinte saída na tela:

```
ALUNO(A)        NOTA
=========       =====
ALINE            9.0
MÁRIO           DEZ
SÉRGIO           4.5
SHIRLEY         7.0
```

8. Letra grande - Elabore um programa para produzir na tela a letra C, de C Progressivo, usando a própria. Se fosse
'C', seria assim:

```
CCCCC
C
C
CCCCC
```

9. Menu - Elabore um programa que mostre o seguinte menu na tela:
Cadastro de Clientes
0 - Fim
1 - Inclui
2 - Altera
3 - Exclui
4 - Consulta
Opção:

10. Pinheiro - Implemente um programa que desenhe um "pinheiro" na tela, similar ao abaixo.

Enriqueça o desenho com outros caracteres, simulando enfeites.

```
          X
         XXX
        XXXXX
       XXXXXXX
      XXXXXXXXX
     XXXXXXXXXXX
    XXXXXXXXXXXXX
   XXXXXXXXXXXXXXX
          XX
          XX
         XXXX
```

Testes condicionais e controle de fluxo em C

Agora que você já estudou a primeira seção de nosso Curso de C, vamos seguir em frente!

Dando continuidade aos conhecimentos básicos, nesta seção vamos ensinar o que são e como usar o teste condicional if else e o teste switch.

Também mostramos outros importantes conceitos da linguagem C, os laços: for, while e do while.

Os testes condicionais e os laços são usados para controlar o fluxo, ordem e a maneira como os programas em C são executados.

Sendo, portanto, conceitos extremamente importantes e utilizados não só em C, mas como toda e qualquer linguagem de programação.

O teste condicional IF ELSE

Se notar bem, em nosso curso online de programação C, todos os nossos programas são rodados uma vez, do início até o fim do código, fazendo sempre a mesma coisa.

Mas você sabe que os programas 'reais' não são assim...se clicar em um botão acontecesse uma coisa, se selecionar algo no menu, outra coisa acontece.

Ou seja, os programas rodam dependendo de como o usuário interage com o aplicativo.
E é isso que vamos estudar nesse tutorial: teste condicional.

- **O que é, para que serve e como usar o teste condicional IF em C**

Sem mais delongas, a sintaxe do teste condicional IF é a seguinte:

```
if( condição )
{
   // código
   // do IF
   // aqui
}
```

O 'if', em inglês, quer dizer 'se', no sentido de condição.
Por exemplo: "If you like C, you will like C++ too" (Se você gosta de C, vai gostar de C++ também).

O que o 'if' faz é testar a 'condição' entre parênteses, e se ela for verdadeira será executado o código entre chaves.

Exemplo de código: Condição sempre verdadeira

```
#include <stdio.h>
int main()
{
    if(1)
    {
       printf("Esse teste condicional é verdadeiro");
    }
}
```

Se você rodar esse programa em C verá que o print sempre aparecerá na tela, pois a condição 1 representa sempre verdadeiro, como você viu no artigo Sistema Binário e Valores lógicos 'true' ou 'false'.

Na verdade, qualquer valor diferente de 0 sempre resultará que o teste condicional é verdadeiro.

Esse 'verdadeiro' ou 'falso' é uma espécie de 'retorno' que a condição nos dá. Por exemplo, vamos testar um código em que o teste condicional nunca é executado:

- **Exemplo de condição: Condição sempre falsa**

```
#include <stdio.h>
int main()
{
    if( 0 )
    {
        printf("Somente programadores verão essa mensagem");
    }

    printf("Hello, World");
}
```

Se você rodar, aparecerá na tela somente o "Hello, World" pois o 0 sempre retorna valor 'false' para o teste condicional IF. Como o resultado é falso, o que está dentro do IF não é executado.

- **O que é, para que serve e como usar o ELSE em C**

A palavra '*else*' em inglês quer dizer 'ou', e ela está totalmente ligada ao IF, que já estudamos.

Se IF significa '*se'* e o ELSE significa '*ou*', você pode pensar e concluir para que serve o ELSE?

O ELSE só existe na presença do IF e só aparece depois que o IF aparecer. A característica principal do ELSE é que ele não recebe nenhuma condição (ou seja, não tem parêntesis depois do ELSE, como tem no IF).

Na verdade ele recebe uma condição, que é implícita. A condição dele é contrária do IF.

Como uma condição é sempre TRUE ou FALSE, no par IF e ELSE um deles vai rodar se a condição for TRUE e o outro só roda se a condição FALSE.

A sintaxe dessa importantíssima dupla é a seguinte:

```
if ( condição )
{
   //se a condição
   //for verdadeira
   //esse código rodará
}
else
{
   //caso a condição
   //seja FALSA
   //é esse código que rodará
}
```

Ou seja, se a condição retornar um valor lógico VERDADEIRO, o IF rodará. Porém, caso a condição retorne um valor lógico FALSO, o ELSE que rodará.

Então, veja o código a seguir, raciocine e tente adivinhar qual print irá rodar, antes de executar o programa:

- **Exemplo de código: rodando o ELSE**

```
#include <stdio.h>

int main()
{
    if( 0 )
    {
        printf("Somente programadores verão essa mensagem");
    }
    else
```

```
    {
        printf("Hello, world!");
    }
}
```

Ora, 0 retorna valor lógico falso, então o IF não será executado.
Sempre que o IF não é executado, o ELSE será. Portanto, veremos a mensagem "Hello, World" na tela.

- **Exercício de C:**

Faça o contrário do último exercício.

Ou seja, que o print "Hello, world!" seja exibido na tela, dentro do bloco do IF e que o bloco do ELSE (com a frase "Somente programadores verão essa mensagem") nunca seja rodado.

Fazendo testes e comparações - operador de igualdade (==), maior (>), menor (<), maior igual (>=), menor igual (<=), de diferença (!=) e de módulo, ou resto da divisão (%)

No artigo passado, ensinamos sobre o uso do teste condicional IF e ELSE, que será extremamente usado por nós em nosso curso, e por você em sua carreira de programador C.

Porém, fizemos testes de condições simples e sem muita utilidade. Vamos aprender agora, em nossa **apostila de C**, a usar alguns operadores que nos permitirão fazer coisas bem interessantes apenas com IF ELSE.

- **Testar e comparar, testar e comparar...a essência da programação**

A primeira vista não damos tanta importância e nem entendemos porque temos tanto que testar e comparar coisas em programação. Pois bem, o curso C Progressivo explica isso melhor pra você, com exemplos reais.

Quando você entra em um site de conteúdo adulto e te perguntam sua idade: se você clicar em 'tenho mais de 18 anos', você entra, se clicar em 'tenho menos de 18 anos', eles te encaminham pro site do Restart.
Pois bem, eles fizeram um **teste**: se fizer isso, acontece isso, se fizer aquilo acontece outra coisa.

As vezes é perguntada nossa data de nascimento, em alguns sites. Informamos os números, eles guardam em alguma variável e depois comparam: se o número for menor que uma data específica, você é de maior. Caso os números que você forneceu sejam maiores que essa data, é porque você ainda é de menor.

Isso foi uma comparação: pegamos um dado (sua data de nascimento) e comparamos com outro (a data de 18 anos atrás).

Durante um jogo, o programa fica o tempo inteiro testando o que o jogador digitou. Se digitou a tecla pra direita, vai pra direita, se foi pra cima, vai pra cima, se não apertar nada, nada ocorre e assim a vida segue.

Se convenceu da importância dos testes e comparações? Então vamos testar e comparar.

- O operador de igualdade em C: ==

Se quisermos comparar uma variável ***x*** com uma variável ***y*** fazemos:
x == y

Sempre que ver isso, leia como se fosse uma pergunta: *x é igual a y?*

Ou seja, se é uma pergunta (comparação), **sempre** retorna um valor booleano. Ou seja, ou é falso ou é verdadeiro.

É comum que alguns estudantes e iniciantes em programação confundam o '=' com o '=='.
O '=' é uma atribuição, estamos atribuindo um valor, estamos dizendo que aquela variável vai receber um valor, e ponto final.

Já o '==' é um teste, queremos saber se o valor é igual ou não.
Ou seja, ele vai retornar TRUE ou FALSE. É como se estivéssemos fazendo uma pergunta, e que estamos esperando uma resposta.

- **Exemplo de código:**

Faça um programa em C que pergunte ao usuário quanto é 1 + 1, usando o teste condicional **IF ELSE**, e que responda se ele acertou ou errou.

```c
#include <stdio.h>

int main(void)
{
    int resultado;

    printf("Quanto eh 1 + 1? ");
    scanf("%d",&resultado);

    if(resultado == 2)
    {
        printf("Parabens, voce nao usa drogas\n");
    }
    else
    {
```

```
        printf("Amigo, pare de fumar isso...\n");
    }
}
```

Ou seja, nós perguntamos o valor de 1+1 e armazenamos na variável inteira ***resultado***.
Depois comparamos esse resultado com 2.

Se o usuário tiver digitado 2, a comparação ***resultado == 2*** retornará valor booleano VERDADEIRO (true) e o ***IF*** ocorre.
Caso o que o usuário digitou não seja 2, a comparação retornará valor booleano FALSO (false) e será o ***ELSE*** que o nosso programa em C irá rodar.

Simples, não? Continuemos...

- **O operador *maior que* em C: >**

Se quisermos saber se uma variável ***x*** é maior que uma variável ***y*** fazemos:
x > y

Caso *x* seja maior que *y*, essa comparação retornará valor lógico VERDADEIRO (true).
Caso *x* seja menor, ou igual, a *y*, essa comparação retornará valor lógico FALSO (false).

Exemplo de código:
Crie um aplicativo em C que pergunte a idade do usuário e diga se ele é maior ou menor de idade, usando o comparador *maior que*.

```
#include <stdio.h>

int main(void)
{
    int idade;

    printf("Digite sua idade: ");
    scanf("%d",&idade);

    if(idade > 17)
```

```
    {
        printf("Voce eh de maior, pode entrar! \n");
    }
    else
    {
        printf("Sinto muito, voce nao pode entrar\n");
    }
}
```

A explicação do código C é bem simples: para ser de maior, você deve ter mais que 17 anos, se tiver, a comparação *idade > 17* retornará TRUE e o ***IF*** irá rodar.
Caso você tenha 17 anos ou menos, você é de menor e o código salta para rodar o ***ELSE***.

Pronto, agora você já pode criar aqueles sistemas de sites que perguntam se você é de maior ou não.
Muito útil essa linguagem de programação C, não?

- **O operador *menor que* em C: <**

Se quisermos saber se uma variável ***x*** é menor que uma variável ***y*** fazemos:
x < y

Caso *x* seja menor que *y*, essa comparação retornará valor lógico VERDADEIRO (true).
Caso *x* seja maior, ou igual, a *y*, essa comparação retornará valor lógico FALSO (false).

Exemplo de código:
Crie um aplicativo em C que pergunte a idade do usuário e diga se ele é maior ou menor de idade, usando o comparador *menor que*.

```
#include <stdio.h>

int main(void)
{
    int idade;
```

```
    printf("Digite sua idade: ");
    scanf("%d",&idade);

    if(idade < 18)
    {
        printf("Sinto muito, voce nao pode entrar\n");
    }
    else
    {
        printf("Voce eh de maior, pode entrar!\n");
    }
}
```

A explicação do código C é o oposto da explicação do exemplo anterior, note a lógica inversa: se você tiver menos que 18 anos, você será menor e o teste ***idade < 18*** retornará valor lógico VERDADEIRO e o ***IF*** será executado. Se você não tem menos que 18 anos, é porque tem mais ora. Então, nosso programa em C executará o ***ELSE***.

- **O operador *maior igual a* em C: >=**

Até o momento fizemos somente a análise do tipo 'maior ou menor'. Porém, as vezes queremos incluir o próprio número na comparação...ou seja, queremos saber se uma variável é maior ou igual que outra.

Se quisermos saber se uma variável ***x*** é maior ou igual que uma variável ***y*** fazemos:
x >= y

Caso *x* seja maior ou igual que *y*, essa comparação retornará valor lógico VERDADEIRO (true).
Caso *x* seja menor a *y*, essa comparação retornará valor lógico FALSO (false).

Exemplo de código:

Crie um aplicativo em C que pergunte a idade do usuário e diga se ele tem que fazer ou não o exame de próstata, usando o operador *maior igual a*. Lembrando que é recomendável fazer o exame a partir dos 45 anos.

```c
#include <stdio.h>

int main(void)
{
    int idade;

    printf("Digite sua idade: ");
    scanf("%d",&idade);

    if(idade >= 45)
    {
        printf("Eh amigo, ja era, tem que fazer o exame da prostata...\n");
    }
    else
    {
        printf("Voce nao precisa fazer o exame de prostata...ainda\n");
    }
}
```

Note quem tem mais de 45 anos é necessário fazer o exame. Porém, quem também TEM, exatos, 45 anos, também precisa fazer!
Também poderíamos ter feito: ***idade > 44***

Porém não fica muito legal o número 44 aparecer. Por quê?
Ora, a informação que temos da idade é 45, então vamos trabalhar com 45. É uma questão de organização apenas, mas que pode facilitar muito a vida de quem lê um código em C.

Por exemplo, suponha que você foi contratado por uma empresa para ser programador C e vai substituir outro programador (que não sabia muito a linguagem C, por isso perdeu o emprego).

Se você bater o olho no código e ver o número **18** logo você associa com maioridade.

Porém, se tiver o número **17** no meio do código fica difícil deduzir o que o antigo programador tentou fazer...aí você vai ter que quebrar a cabeça pra entender o código dele.

Mas como você está cursando o C Progressivo, você será treinado para ser um programador organizado também. Vamos continuar...

- **O operador *menor igual a* em C: <=**

Se quisermos saber se uma variável ***x*** é menor ou igual que uma variável ***y*** fazemos:
x <= y

Caso *x* seja menor ou igual que *y*, essa comparação retornará valor lógico VERDADEIRO (true).
Caso *x* seja maior a *y*, essa comparação retornará valor lógico FALSO (false).

- **Exemplo de código:**

Crie um aplicativo em C que pergunte a idade do usuário e diga se ele tem que fazer ou não o exame de próstata, usando o operador *menor igual a*. Lembrando que é recomendável fazer o exame a partir dos 45 anos.

```
#include <stdio.h>

int main(void)
{
   int idade;

   printf("Digite sua idade: ");
   scanf("%d",&idade);

   if(idade <= 44)
   {
      printf("Voce nao precisa fazer o exame de prostata...ainda\n");
   }
   else
   {
      printf("Eh amigo, ja era, tem que fazer o exame da prostata...\n");
   }
}
```

Agora fizemos o contrário. Comparamos usando a expressão ***idade <= 44*** para verificar que o usuário não tem 45 anos ou mais.

Analisando esse código e o do exemplo passado, qual você usaria para criar tal programa?
Arraste o mouse aqui para ver a resposta ->

- **O operador *diferente de* em C: !=**

Se quisermos saber se uma variável ***x*** é diferente de uma variável ***y*** fazemos:
x != y

Caso *x* seja diferente de *y*, esse teste retorna valor VERDADEIRO.
Caso *x* seja igual a *y*, esse teste retorna valor FALSO.

- **Exemplo de código:**

Crie um programa em C que pergunte um número de 1 até 10 ao usuário, e faça com que ele tente acertar o número secreto. No meu caso, o número secreto é 7.

```
#include <stdio.h>

int main(void)
{
   int numero;

   printf("Adivinha a senha, entre 1 e 10: ");
   scanf("%d",&numero);

   if(numero != 7)
   {
      printf("Errado! Saia da Matrix!\n");
   }
   else
   {
      printf("Parabens, voce entrou no sistema\n");
   }
}
```

A explicação é simples: enquanto o usuário digitar um número diferente de 7, o programa entra no ***IF*** e diz que o usuário errou a senha.

Se ele não digitar um número de 7, é porque ele digitou o 7, ora! Então, a comparação ***numero != 7*** retorna falso e o ***ELSE*** que será executado.

- **O operador *módulo ou resto da divisão* em C: %**

Resto da divisão é o valor que sobra/resta da divisão para que o quociente permaneça inteiro.

O símbolo de resto da divisão é %.

Veja, o que está em vermelho é o resto inteiro da divisão:
5 % 2 = 2*2 + 1
7 % 3 = 3*2 + 1
7 % 4 = 1*4 + 3

8 % 3 = 2. Pois: 8 = 3*2 + 2

Em matemática chamamos de 'mod', de módulo: 5 mod 3 = 2

No Brasil, é aquele 'resto' que deixávamos lá embaixo nas continhas do colégio:

Sem dúvidas, a maior utilidade desse operador é saber a multiplicidade de um número (ou seja, se é múltiplo de 2, 3, 5, 7, etc).
Por exemplo, um número par é aquele que, quando dividido por 2, deixa resto nulo.

- **Exemplo de código:**

Crie um programa em C que pergunte um número ao usuário e diga se ele é par. Use o operador módulo ou resto da divisão ***%.***

```
#include <stdio.h>

int main(void)
{
   int numero;

   printf("Digite um inteiro: ");
   scanf("%d",&numero);

   if( (numero % 2) == 0)
   {
      printf("Eh par\n");
   }
   else
   {
      printf("Nao eh par\n");
   }
}
```

Em programação, para saber se um número é par, escrevemos: ***numero % 2 == 0***
Leia: se o resto da divisão de 'numero' por 2 deixar resto 0, retorne valor TRUE.

- **Erro comum de iniciantes em programação C**

Praticamente todos os programadores, quando estão iniciando seus estudos, cometem o erro de confundir o operador de ***atribuição*** com o operador de ***comparação de igualdade***.

Por exemplo, qual a diferença entre:
x = 1
x == 1
?

A resposta é simples: no primeiro exemplo estamos atribuindo, no segundo estamos comparando.

Leia assim:
x = 1: 'Hey C, faça com que a variável x receba o valor 1, atribua 1 ao x'
x == 1: "Hey C, o valor armazenado na variável x é 1'?

Ou seja, na comparação há o retorno de valor lógico VERDADEIRO ou FALSO.
Na atribuição não, simplesmente atribuímos um valor e essa operação retorna sempre true.

- **Exercícios sobre operadores de comparação em C:**

1. Faça o programa do exemplo do ***!=*** , mas com o operador **==**
2. Faça o programa do exame de próstata usando o operador **<**
3. Faça o programa do exame de próstata usando o operador **>**
4. Faça o programa do teste de maioridade usando o operador **<=**
5. Faça o programa do teste de maioridade usando o operador >**=**
6. Faça um programa que pergunte um número ao usuário e diga se ele é ímpar. Use o operador ***%***.

Questões sobre IF e ELSE

Agora que já ensinamos o que é e como usar o teste condicional IF-ELSE e operadores relacionais em linguagem C(para fazer testes e comparações), agora vamos propor alguns exercícios, em nossa apostila de C, e mostrar suas soluções, com códigos comentados, para você fixar seus conhecimentos nesses conceitos tão importantes em C.

Lembramos que esses exercícios fazem parte do curso, além de ensinarmos técnicas de programação e assuntos importantes através deles.

- **Exercícios sobre IF e ELSE em C**

Embora venhamos a mostrar a solução dos exercícios, **não olhe a solução antes de tentar**.
É simples: você só aprende se tentar. Se estiver só lendo, copiando e colando os códigos no Code::Blocks, você nunca aprenderá. Tente, tente de novo, depois tente mais um pouco.
Só então olhe a solução.

0. Faça um programa que peça dois números ao usuário e mostre qual o maior e qual o menor

1. Faça um programa que receba três inteiros e diga qual deles é o maior e qual o menor. Consegue criar mais de uma solução?

2. Escreva um programa em C que recebe um inteiro e diga se é par ou ímpar.
Use o operador matemático % (resto da divisão ou módulo) e o teste condicional if.

3. Escreva um programa que pergunte o raio de uma circunferência, e sem seguida mostre o diâmetro, comprimento e área da circunferência.

4. Para doar sangue é necessário ter entre 18 e 67 anos. Faça um aplicativo na linguagem C que pergunte a idade de uma pessoa e diga se ela pode doar sangue ou não. Use alguns dos operadores lógicos OU (||) e E (&&).

5. Escreva um programa que pergunte o dia, mês e ano do aniversário de uma pessoa e diga se a data é válida ou não. Caso não seja, diga o motivo. Suponha que todos os meses tem 31 dias e que estejamos no ano de 2013.

Desafio 1. Crie um programa em C que peça um número ao usuário e armazene ele na variável *x*. Depois peça outro número e armazene na variável *y*.
Mostre esses números. Em seguida, faça com que *x* passe a ter o valor de *y*, e que *y* passe a ter o valor de *x*.
Dica: você vai precisar usar outra variável.

Desafio 2. Escreva um programa que pede os coeficientes de uma equação do segundo grau e exibe as raízes da equação, sejam elas reais ou complexas.

Desafio 3. Crie um programa em C que recebe uma nota (entre 0.0 e 10.0) e checa se você passou direto, ficou de recuperação ou foi reprovado na matéria.
A regra é a seguinte:
Nota 7 ou mais: passou direto
Entre 4 e 7: tem direito de fazer uma prova de recuperação
Abaixo de 4: reprovado direto

- Questões resolvidas sobre IF ELSE

0. Faça um programa que peça dois números ao usuário e mostre qual o maior e qual o menor

Vamos declarar dois números inteiros: x e y
Pedimos para o usuário preencher o valor de x, em seguida o de y.

Primeiro vamos checar se x é maior que y através do operador maior igual que, como teste para o teste condicional IF. Caso seja verdade, dizemos que x é maior. Caso seja falso, o teste cai no ELSE e diz que y é maior.

```
#include <stdio.h>

int main()
{
    int x, y;
```

```
    printf("Digite o numero 1: ");
    scanf("%d", &x);

    printf("Digite o numero 2: ");
    scanf("%d", &y);

    if(x > y)
        printf("Maior: x = %d", x);
    else
        printf("Maior: y = %d", y);

}
```

Note que, na linha abaixo do IF e na linha abaixo do ELSE nós não colocamos o código entre chaves {}.
Só podemos fazer isso se o código abaixo do IF ou do ELSE possuam SOMENTE UMA LINHA.
Isso mesmo, o C só irá interpretar a linha imediatamente abaixo do IF ou do ELSE como fazendo parte desses testes condicionais.

Ou técnica muito importante, que ensinaremos aqui nesse tutorial de C, é o uso de uma terceira variável que irá receber o valor do maior número entre x e y.
Vejam só:

```
#include <stdio.h>

int main()
{
    int x,
        y,
        maior;
    printf("Digite o numero 1: ");
    scanf("%d", &x);

    printf("Digite o numero 2: ");
    scanf("%d", &y);

    if(x > y)
        maior = x;
    else
        maior = y;
```

```
    printf("O maior numero eh: %d", maior);


}
```

1. Faça um programa que receba três inteiros e diga qual deles é o maior e qual o menor. Consegue criar mais de uma solução?

Vamos declarar três variáveis: num1, num2 e num3.
Primeiro vamos checar se num1 é maior que num2. Se for, dentro do IF fazemos outro teste pra saber se num1 é maior que num3 também, se for, é porque num1 é o número maior.

Aqui notamos um detalhe importante e muitíssimo usado em programação: if else aninhados.
Ou seja, aninhar é colocar um dentro do outro. No caso, colocamos outro teste if-else dentro de um teste if.

No caso, primeiro testamos se num1 é maior que num2. Somente se num1 for maior que num2 é que vamos testar se num1 é maior que num3.
Se num1 não for maior que num2, o teste pra saber se num1 é maior que num3 nem acontece.

É importante você identar tudo, para saber a qual IF cada ELSE pertence. É extremamente importante que você tenha essa organização para não criar confusão.

Pois bem, testamos para saber se num1 é maior que num2.
Aqui, outro caso pode acontecer: num1 é maior que num2, mas num1 não é maior que num3.
Raciocine comigo: se num1 é maior que num2 e num3 é maior que num1, então num3 também é maior que num2 e num3 é o maior número.
Pronto, aqui terminamos o primeiro IF, em que num1 é sempre maior que num2.

Agora, pode ser que num2 seja maior que num1. Isso cai no ELSE maior.
Ok, aqui sabemos que num2 é maior que num1. Fazemos outro teste condicional IF ELSE dentro desse else maior para saber se num2 é maior que num3 também, se for, mostramos que num3 é o maior.

Porém, se num2 não for maior que num3, é óbvio que num3 é maior que num2.
Como num2 é maior que num1 e se num3 é maior que num2, obviamente num3 também é maior que num1, e mostramos que num3 é o maior número.

O raciocínio para descobrir o menor número é parecido, porém, ao invés de usar o operador maior que (>), se usa o operador menor que (<). Tente!

```c
#include <stdio.h>

int main()
{
    int num1,
        num2,
        num3;

    printf("Digite o numero 1: ");
    scanf("%d", &num1);

    printf("Digite o numero 2: ");
    scanf("%d", &num2);

    printf("Digite o numero 3: ");
    scanf("%d", &num3);


    // Lógica para descobrir o maior número
    if (num1 > num2)
    {
        if(num1 > num3)
        {
            printf("O maior numero e: %d\n", num1);
        }
        else
        {
            printf("O maior numero e: %d\n", num3);
        }
    }
    else
    {
        if(num2 > num3)
        {
            printf("O maior numero e: %d\n", num2);
        }
        else
        {
            printf("O maior numero e: %d\n", num3);
        }
    }
```

```
    // Lógica para descobrir o menor número
    if(num1 < num2)
    {
        if(num1 < num3)
        {
            printf("O menor numero e: %d", num1);
        }
        else
        {
            printf("O menor numero e: %d", num3);
        }
    }
    else
    {
        if(num2 < num3)
        {
            printf("O menor numero e: %d", num2);
        }
        else
        {
            printf("O menor numero e: %d\n", num3);
        }
    }


}
```

Uma outra maneira de fazer seria usando duas outras variáveis para armazenar o maior e o menor número: 'menor' e 'maior'.
Ao final, exibimos apenas um printf mostrando o valor dessas variáveis.
Veja como ficou nosso código em C:

```
#include <stdio.h>

int main()
{
    int num1,
        num2,
        num3,
        maior,
        menor;

    printf("Digite o numero 1: ");
```

```
    scanf("%d", &num1);

    printf("Digite o numero 2: ");
    scanf("%d", &num2);

    printf("Digite o numero 3: ");
    scanf("%d", &num3);


    // Lógica para descobrir o maior número
    if (num1 > num2)
    {
        if(num1 > num3)
            maior = num1;
        else
            maior = num3;
    }
    else
    {
        if(num2 > num3)
            maior = num2;
        else
            maior = num3;
    }

    // Lógica para descobrir o menor número
    if(num1 < num2)
    {
        if(num1 < num3)
            menor = num1;
        else
            menor = num3;
    }
    else
    {
        if(num2 < num3)
            menor = num2;
        else
            menor = num3;
    }

    printf("O maior numero e: %d\n", maior);
    printf("O menor numero e: %d\n", menor);

}
```

2. Escreva um programa em C que recebe um inteiro e diga se é par ou ímpar.
Use o operador matemático % (resto da divisão ou módulo) e o teste condicional if.

Essa questão é bem simples e já foi resolvida de uma maneira em nosso artigo sobre operadores de comparação e teste em C.

Lá mostramos que o IF identifica se o número é par, se não for, vai pro ELSE.
Vamos fazer o contrário aqui: se cair no IF é porque o número é ímpar. Se cair no ELSE, é porque é par.

Pra saber se um número é par, é só testar se o o resto da divisão desse número por 2 é 0.
É fácil ver que, para checar se um número é ímpar, basta testar se o resto da divisão desse número por 2 é 1. Assim, nosso simples aplicativo em C fica:

```
#include <stdio.h>

int main()
{
    int num;

    printf("Digite um numero: ");
    scanf("%d", &num);

    if( num%2 == 1 )
        printf("%d e ímpar\n", num);
    else
        printf("%d e par\n", num);
}
```

3. Escreva um programa que pergunte o raio de uma circunferência, e sem seguida mostre o diâmetro, comprimento e área da circunferência.

Nas outras questões vínhamos usando apenas números inteiros em nossas aplicaçõe em C. Porém, nada impede você de usar float ou double, irá funcionar do mesmo jeito.

Mas nessa questão temos que usar float, pois vamos tratar de números com casas decimais, como o pi.
Fora isso, a questão é bem óbvia e simples. Basta aplicar as fórmulas matemáticas:

```c
#include <stdio.h>

int main()
{
    float raio,
          pi = 3.14,
          diametro,
          comprimento,
          area;

    printf("Digite o raio: ");
    scanf("%f", &raio);

    diametro = 2 * raio;
    comprimento = 2 * pi * raio;
    area = pi * raio * raio;

    printf("Diametro %.2f\n", diametro);
    printf("Comprimento %.2f\n", comprimento);
    printf("Area %.2f\n", area);
}
```

Deixamos para você o exercício de criar um programa em C, igual ao passado, porém usando somente a variável 'raio'.

4. Para doar sangue é necessário ter entre 18 e 67 anos. Faça um aplicativo na linguagem C que pergunte a idade de uma pessoa e diga se ela pode doar sangue ou não. Use alguns dos operadores lógicos OU (||) e E (&&).

Vamos agora aliar nossos conhecimentos sobre o teste condicional IF ELSE com os operadores lógicos, tudo já foi devidamente ensinado aqui em nossa apostila online de C.

Note que, para doar sangue é necessário ter 18 anos ou mais, E TAMBÉM é necessário ter 67 anos ou menos.
Ou seja, podemos fazer usando o operador lógico &&: (ter 18 anos ou mais) && (ter 67 anos ou menos).

```c
#include <stdio.h>

int main()
{
```

```
    int idade;

    printf("Digite sua idade: ");
    scanf("%d", &idade);

    if( (idade >= 18) && (idade <=67))
        printf("Voce pode doar sangue\n");
    else
        printf("Voce nao pode doar sangue\n");
}
```

No código passado mostramos o que era necessário para doar sangue e cair no IF.

Agora vamos fazer o contrário: vamos fazer o teste no IF para mostrar a mensagem de que o usuário NÃO pode doar sangue.

Para NÃO PODER doar sangue o usuário deve ter menos de 18 anos OU mais de 67 anos, concorda?
Então, usaremos o operador lógico ||: (ter menos de 18 anos) || (ter mais de 67 anos)

```
#include <stdio.h>

int main()
{
    int idade;

    printf("Digite sua idade: ");
    scanf("%d", &idade);

    if( (idade < 18) || (idade > 67))
        printf("Voce nao pode doar sangue\n");
    else
        printf("Voce pode doar sangue\n");
}
```

5. Escreva um programa que pergunte o dia, mês e ano do aniversário de uma pessoa e diga se a data é válida ou não. Caso não seja, diga o motivo. Suponha que todos os meses tem 31 dias e que estejamos no ano de 2013.

Um dia válido é aquele que está entre 1 e 31. Um número abaixo de 1 ou acima de 31 será considerado inválido.

Um mês válido é aquele que está entre 1 e 12. Um número abaixo de 1 ou acima de 12 será considerado inválido.

Um ano de nascimento válido é aquele que está abaixo do ano atual. Como estamos em 2013, se alguém fornecer dados de ano 2014 ou outro ano acima disso, será considerado inválido.

Vamos mostrar, mais uma vez, nesse tutorial mais um exemplo de if-else aninhados.

Primeiro fazemos o teste para saber se a idade é inválida.
Se for, o IF roda e acaba o programa.

Caso a idade seja válida, nosso programa em C vai para o ELSE.
No ELSE, testamos se o mês é inválido. Se for, mostra a mensagem de erro e programa acaba aí.
Se o mês for válido, testamos se o ano é invalido. Se for, mostra a mensagem de erro do ano e o programa em C acaba.
Se o ano for inválido, mostra a mensagem que o ano é inválido.
Porém, se o ano também for válido, mostra a mensagem que a data é válida e se encerra nossa aplicação C.

Note que a data só é válida se o dia for válido. Dentro dessa condição, o mês deve ser válido também.
E dentro da condição de verdadeiro do mes, o ano deve ser válido também.
Assim, a mensagem de data correta só é exibida se os 3 testes (dia, mes e ano) validarem a data.

```
#include <stdio.h>

int main()
{
    int dia,
        mes,
        ano;

    printf("Dia: ");
```

```
    scanf("%d", &dia);

    printf("Mes: ");
    scanf("%d", &mes);

    printf("Ano: ");
    scanf("%d", &ano);


    if((dia < 1) || (dia > 31))
        printf("Dia invalido\n");
    else //se o dia for válido
        if( (mes < 1) || (mes > 12) )
            printf("Mes invalido\n");
        else // além do dia, o mês for válido
            if( ano > 2013 )
                printf("Ano invalido\n");
            else //se além do dia e mês, o ano for válido
                printf("Data valida\n");


}
```

Desafio 1. Crie um programa em C que peça um número ao usuário e armazene ele na variável x. Depois peça outro número e armazene na variável y.
Mostre esses números. Em seguida, faça com que x passe a ter o valor de y, e que y passe a ter o valor de x.
Dica: você vai precisar usar outra variável.

O que vamos fazer é aqui é simplesmente trocar os valores de duas variáveis: x e y
Parece simples e bobo, mas não é óbvio e esse algoritmo é muito importante, e você verá ele aqui no curso C Progressivo várias vezes, e durante sua carreira como programador C.

Se queremos trocar o valor de x pelo valor de y, basta fazermos:
x = y;

Agora, queremos fazer com que y receba o valor de x:
y = x;

Correto? Não, errado.
y não terá o valor antigo de x, pois agora x tem um novo valor (y). Então, como resolver?

Vamos usar uma variável de apoio, também chamada de variável temporária. Ela servirá para armazenar o valor antigo de x. Assim, y pode pegar o valor antigo de x.

```c
#include <stdio.h>

int main()
{
    int x,
        y,
        tmp;

    printf("x =  ");
    scanf("%d", &x);

    printf("y =  ");
    scanf("%d", &y);

    printf("\nTrocando os valores de x e y...\n");

    // É importante que entenda e decore as seguintes linhas
    // Elas são muito importantes e usadas em programação C
    tmp = x;
    x = y ;
    y = tmp;

    printf("\nx = %d \ny = %d\n",x,y);

}
```

Desafio 2.
Escreva um programa na linguagem C que pede os coeficientes de uma equação do segundo grau e exibe as raízes da equação, sejam elas reais ou complexas.

Passo 1:
A primeira parte do programa recebe os três coeficientes da equação, que são 'a', 'b' e 'c' e serão representados pelo tipo float.

Passo 2:
Uma equação do 2o grau só é válida se 'a' for diferente de 0, então, se for igual a 0 o programa deverá terminar.
Ou seja, nosso programa irá acontece dentro do 'if' que checa que 'a' é diferente de 0.

Passo 3:
Determinando o valor de delta: delta = b^2 - 4ac
Determinando a raiz quadrada de delta: sqrtdelta = sqrt(delta);

Onde *sqrt* significa 'square root', ou raiz quadrada, em inglês.
Em C, calculamos a raiz quadrada do número x usando a função *sqrt()*, que está na biblioteca *math.h*. Logo, precisamos importar essa biblioteca no começo do programa.

Passo 4:
Se delta for maior ou igual a zero, as raízes são dadas por:
raiz1 = (-b + sqrtdelta) / 2a
raiz2 = (-b - sqrtdelta) / 2a

Passo 5:
Se delta for menor que zero, suas raízes serão complexas e as raízes serão da forma:
raiz1 = (-b + i.sqrt(-delta))/2a
raiz2 = (-b - i.sqrt(-delta))/2a

Formatei a saída da seguinte forma, para ficar mais legível, que é a mesma coisa das equações anteriores:
raiz1 = (-b)/2a + i.sqrt(-delta)/2a
raiz2 = (-b)/2a - i.sqrt(-delta)/2a

Assim, chegamos ao resultado final de nosso programa em C:

```c
#include <stdio.h>
#include <math.h>
int main()
{
    float a, b, c,    //coeficientes
```

```
  delta,      //delta
  sqrtdelta,  //raiz quadrada de delta
raiz1,raiz2;  //raízes

//Passo 1: Recebendo os coeficientes
printf("Equação do 2o grau: ax² + bx + cx = 0\n");

printf("Entre com o valor de a: ");
scanf("%f", &a);

printf("Entre com o valor de b: ");
scanf("%f", &b);

printf("Entre com o valor de c: ");
scanf("%f", &c);


//Passo 2: Checando se a equação é válida
if(a != 0)
{

//Passo 3: recebendo o valor de delta e calculando sua raiz quadrada
    delta = (b*b) - (4*a*c);
    sqrtdelta = sqrt(delta);

//Passo 4: se a raiz de delta for maior que 0, as raízes são reais
    if(delta >= 0)
    {
        raiz1 = (-b + sqrtdelta)/(2*a);
        raiz2 = (-b - sqrtdelta)/(2*a);
        printf("Raízes: %.2f e %.2f", raiz1, raiz2);
    }
//Passo 5: se delta for menor que 0, as raízes serão complexas
    else
    {
        delta = -delta;
        sqrtdelta = sqrt(delta);
        printf("Raíz 1: %.2f + i.%.2f\n", (-b)/(2*a), (sqrtdelta)/(2*a));
        printf("Raíz 2: %.2f - i.%.2f\n", (-b)/(2*a), (sqrtdelta)/(2*a));

    }

}
else
    printf("Coeficiente 'a' inválido. Não é uma equação do 2o grau");
```

}

Desafio 3. Crie um programa em C que recebe uma nota (entre 0.0 e 10.0) e checa se você passou direto, ficou de recuperação ou foi reprovado na matéria.

A regra é a seguinte:
Nota 7 ou mais: passou direto
Entre 4 e 7: tem direito de fazer uma prova de recuperação
Abaixo de 4: reprovado direto

Vamos fazer isso usando condições, testando com if e else o que foi digitado pelo usuário. No caso, a nota.
Vamos colocar if dentro de if, essa técnica se chama aninhar (nested).

Vou dar uma importante dica que você vai usar por toda sua vida de programador, que é uma das maiores lições que sei e posso passar:
"Um problema difícil nada mais é que uma série de pequenos problemas fáceis."

E qual pra transformar um problema difícil em fáceis?
Usando a técnica de Jack, o estripador: por partes. Quebre o programa.

Todo e qualquer projeto, desde esse que vou passar até os da NASA, são divididos em área, partes...na criação de um jogo, tem gente especializada até no desenho das árvores, no barulho das armas e se o eco vai ser realista.

Ok, vamos lá.
Criar um programa que recebe uma nota do usuário você já sabe, é só usar a função scanf.

Mas o nosso curso vai ser focado nas boas práticas, vamos focar em criar aplicações robustas.
O que é isso?
Vamos minimizar o máximo possível os possíveis problemas que possam aparecer quando usuário for usar o problema.

Parte 1: Tendo a certeza que o usuário vai digitar uma nota válida
Você pensou nisso? E se o energúmeno digitar 11 ou -1.1 ?
Não pode.

O primeiro passo da nossa aplicação é ter a certeza que ele vai digitar uma nota ENTRE 0.0 e 10.0!

Se for 0.0 até 10.0, ok, e se não for? Acaba o programa.
Fica assim:

```
printf("Digite sua nota [0.0 - 10.0]: " );
scanf("%f", &nota);

if( (nota <= 10.0) && (nota >= 0.0) )
{
   printf("Nota válida\n");
}
else
{
   printf("Nota inválida, fechando aplicativo");
}
```

O que fizemos foi certificar que a nota é menor que 10.0 E maior 0.0 !
Os dois casos precisam ser satisfeitos, por isso o operador lógico &&.

Teste. Coloque 10.1, 10.5, 21.12, -19, -0.000001, "c progressivo".
Você tem que fazer muito isso: testar.
Nas grandes empresas tem gente contratada só pra isso: testar (estagiários ou bolsistas).

Eliminada as possibilidades de notas absurdas, prossigamos.

Parte 2:
Checando se passou direto
Vamos colocar mais um if else que vai checar se o aluno passou direto ou não.
Por hora, esqueça o código feito.

Como seria esse código pra testar se ele passou direto? Fácil:

```
if( nota >= 7.0 )
{
   printf("Parabens, voce passou direto. Voce deve ser programador C, por acaso estuda pela apostila C Progressivo ?\n");
}
else
{
        printf("Nao passou direto");
}
```

Vamos colocar esse trecho de código abaixo daquele print que diz 'nota válida'.
Então, vamos pensar como o C pensa:

Primeiro ele recebe o resultado.
Checa se está entre 0.0 e 10.0. Se não está, pula pro else e terminou o programa.

Se estiver entre 0.0 e 10.0, diz que a nota é válida e depois faz outro teste em outro if, pra checar se a nota é maior que 7.0, se for diz passou direto, se não for diz que não passou direto.

O código ficaria assim:

```
if( (nota <= 10.0) && (nota >= 0.0) )
{
   printf("Nota válida");

   if( nota >= 7.0 )
   {
      printf("Parabens, voce passou direto. Voce deve ser programador C, por acaso estuda pela apostila C Progressivo ?\n");
   }
   else
   {
      printf("Não passou direto\n");
   }

}
else
{
   printf("Nota inválida, fechando aplicativo");
}
```

Teste.
Note que o C é um professor super rigoroso. Se tirar 6.99 não passa direto!

Parte 3:
Checando se você ainda há esperanças
Bom, agora vamos checar se o aluno ainda pode fazer recuperação, ou seja, se a nota está entre 4.0 e é menor que 7.0.

Caso seja exatamente 7.0, passará automaticamente.
O código pra essa parte é o seguinte:

```
if( nota >= 4.0 )
{
    printf("Vai ter que fazer recuperacao");
}
else
{
    printf("Reprovado. Estude C que voce ficara mais inteligente");
}
```

Note que se não for maior ou igual a 4.0, é menor, então vai pro else.
Ora, se é menor, foi reprovado. Então o else manda mensagem de reprovação.
A pergunta é, onde vamos inserir esse código?
No else do trecho 'Não passou direto'

Então, o código completo em C fica:

```
#include <stdio.h>

int main()
{
        float nota; //vai armazenar a nota

        printf("Digite sua nota [0.0 - 10.0]: " );
        scanf("%f", &nota);

        if( (nota <= 10.0) && (nota >= 0.0) )
        {
            if( nota >= 7.0 )
            {
                printf("Parabens, vocc passou direto. Voce deve ser programador C\n");
            }
            else
            {

                if( nota >= 4.0 )
                {
                    printf("Voce vai para a recuperacao\n");
                }
```

```
            else
            {
                printf("Reprovado. Estude C que voce ficara mais inteligente\
n");
            }
        }

    }
    else
    {
        printf("Nota inválida, fechando o programa C\n");
    }

}
```

Operadores de Incremento (++) e Decremento (--) - Diferença entre a=b++ e a=++b

Antes de iniciarmos nossos estudos sobre os laços, o curso C Progressivo irá apresentar algumas ferramentas que serão úteis e bastante utilizadas.

São operadores matemáticos que facilitarão nossas vidas de programador C, e que iremos usar diversas vezes em nossa apostila online.

- **Contando em linguagem C**

Em muitas aplicações C precisamos contar, ou controlar coisas:

•quantas teclas foram digitadas
•quantas vidas você ainda tem em jogo
•quanto tempo se passou
•quantos visitantes o site C Progressivo recebeu
•etc.

Até agora, temos contado de um em um. Imagine, gerenciar e contar um sistema de cadastro com milhões de inscritos, como o do Enem?

Ainda bem que você é programador C e não se assustou, pois sabe que não vai fazer isso, e sim seu escravo: o computador.
Lembre-se: a relação do programador C com a máquina não é de usuário, e sim de mestre para escravo. Com C, você domina tudo o que ocorre com seu computador.

Então, boa parte das contas são feitas utilizando os 'laços', que ensinaremos bem em breve e vai utilizar para sempre.
Com ele, você vai fazer com que o computador faça o tanto de contas que quiser...10, 20, mil, 1 milhão...o Google realiza trilhões por dia.
Você pode até fazer um looping infinito e travar seu computador. Uns chamam de hackear, eu prefiro chamar de estudo e vou mostrar como fazer.

Esses laços, porém, tem que ser incrementado. Geralmente eles tem um início, uma condição e um fim.

Como esse fim é atingido? Através de uma ou mais variável que vão crescendo ou diminuindo, ou através de expressões lógicas.
Quando se atinge um certo ponto, o laço vai parar.

É aí que entra os operadores de incremento e decremento.

- **Somando e Subtraindo variáveis em C**

O incremento em uma unidade quer dizer:
a = a + 1

a = a + 1, em C, é algo totalmente diferente da Matemática convencional.
Vamos ver na prática, que depois explico.

Façamos isso em um aplicativo em C:
a=1;
a = a + 1;
Logo após atribuir 1 ao 'a', imprima, veja o resultado e tente descobrir o que aconteceu.
Substitua por: a = a + 2;
Ou a = a + 3;

```
#include <stdio.h>

int main()
{
	int a=1;

	printf("a = %d\n", a);
	a = a + 1;
	printf("a = %d\n", a);

}
```

Hackers são pessoas que descobrem as coisas sozinhas (e não criminosos ou vândalos, como é difundido), fazem isso.
Testam, pensam e descobrem como as coisas funcionam.

Explicação:
Quando fazemos a =

é porque vamos atribuir um valor ao 'a', um novo valor, INDEPENDENTE do que ele tinha antes.
E qual o valor que vamos atribuir? 'a + 1'!
Ou seja, o 'a' vai ser agora o seu antigo valor mais 1!

Se a=2, e fazemos a = a + 3, o que estamos fazendo?
Estamos dizendo que o 'a' vai mudar, vai ser seu antigo valor (2), mais 3! Ou seja, agora a=5
Faz sentido, não?

a++ e a--
Usaremos muito, mas MUITO mesmo o incremento e o decremento de unidade:
a = a + 1;
a = a - 1;

Porém, é chato ou da preguiça ficar escrevendo a=a+1 e a=a-1 o tempo inteiro.
Então, inventaram atalhos para isso!
a = a + 1 pode ser representado por a++ ou ++a
a = a - 1 pode ser representado por a-- ou --a

Existe uma diferença entre a-- e --a, ou entre a++ e ++a, que vamos explicar no próximo tópico.
E como já dizia o filósofo: só creio compilando e vendo.

```c
#include <stdio.h>

int main()
{       int a=1,
           b=1;

        printf("Valor inicial de a = %d\n", a);
        printf("Valor inicial de b = %d\n", b);

        printf("\nIncrementando: a++\n");
        a++;

        printf("Decrementando: b--\n");
        b--;
```

```
        printf("\nAgora a = %d\n", a);
        printf("Agora b = %d\n",b);

}
```

- **Diferença entre os operadores entre a=++b e a=b++**

Já vimos o uso dos operadores de decremento e incremento, em C.

Você ja estudou e testou que usar isoladamente:
a++ e ++a não surte efeito.

Porém, surte na hora da atribuição.

Vamos lá, ao nosso 'teste hacker':
Seja b=2;
Só olhando e raciocinando, você seria capaz de descobrir quanto seria 'a' em cada caso:
a = b++;
a = ++b;

Explicações
1. a = b++

Isso é apenas um atalho em C, uma forma mais simples de escrever as seguintes linhas:
a = b;
b++;

2. a = ++b
Analogamente, é o atalho que usamos em C para representar as seguintes linhas de código:
++b;
a = b;

Em ambos os casos, 'b' é incrementado. A diferença é que 'a' recebe o valor de 'b' antes do incremento em um e depois do incremento em outro. Não

precisa decorar nada, é só analisar a lógica das coisas e verá que tudo faz sentido na linguagem C.
Veja:

No primeiro exemplo: a = b++
O que vem primeiro, 'b' ou incremento? Primeiro o 'b'. Depois o de incremento ocorre, '++'.
Ou seja, 'a' vai receber o valor de 'b' primeiro, então a=2
Só depois que 'b' vai ser incrementado e vai se tornar b=3.

No segundo exemplo: a = ++b
O que vem primeiro? 'b' ou incremento?
Primeiro o incremento, '++', depois que aparece o 'b', então só depois acontece a atribuição..
Assim, primeiro ocorre o incremente, então b=3.
Só depois é que esse valor é atribuído para a.

Como diria o véio deitado: compilais e verás a verdade.
Eis o código:

```c
#include <stdio.h>
int main()
{
	int a,
	    b=1;

	printf("b = %d\n",b);
	printf("a = b++\n");

	a = b++;

	printf("\nAgora: \na = %d\n",a);

	printf("b = %d\n",b);
	printf("\na = ++b\n");

	a = ++b;

	printf("\nAgora: a = %d",a);
}
```

O que é e como usar o laço WHILE em C

Com o teste condicional IF ELSE, explicado aqui na **apostila C Progressivo**, você passou a ter mais controle sobre suas aplicações em C, pois agora você pode escolher o que será rodado em seu programa, dependendo dos valores lógicos que esse teste condicional faz.

Vamos agora introduzir outro importante assunto em linguagem de programação C: laços. Também conhecidos por loopings.

Com laços e testes, você passará a ter total controle do fluxo em seus códigos.

- **Repetir e repetir: o que são e para que servem os laços em C**

Repetir é uma das coisas mais importantes na programação. Na verdade, elas não se repetem indefinidamente, elas se repetem dependendo dos testes que fazem.

Por exemplo, seu computador faz várias repetições para saber se você está conectado a internet.
Se esse teste der resultado positivo, nada acontece.
Porém, se ele testar e ver que a conexão não existe mais, ele manda uma mensagem dizendo que você perdeu a conexão.

Fazer testes você já está careca de saber.
Porém, como faria todos esses testes? Colocaria um if else a cada segundo, durante 1h?
E se o usuário passar 2h conectado? Ou 1 dia?

E se você fosse contratado para criar um programa em C que diz se cada aluno em uma sala de aula passou.
Se essa sala tem 30 alunos, você vai criar 30 testes condicionais?
E se a sala tiver 60 alunos? Ou se você for contratado para testar as notas de todos os 2mil alunos de uma escola?
2mil blocos de IF ELSE?

Óbvio que não. Lembre-se: você programa em C, então você é quem manda no seu computador.

Você fará sua máquina repetir essas coisas automaticamente.

- **O laço while: definição e sintaxe**

While, em inglês, significa 'enquanto'.
A função do laço while é repetir um determinado trecho ENQUANTO uma determinada condição for verdadeira.

Por exemplo, enquanto (while) o personagem tiver *life*, o jogo deve continuar.
Enquanto (while) a mp3 não terminar, ela deve continuar tocando.
Enquanto (while) um jogador não ganhar, perder ou der empate, o jogo da velha deve continuar rodando.
Enquanto o usuário não digitar 0, seu programa deve continuar rodando.

E poderíamos dar milhões de exemplos do quanto os laços, como o *while*, são importantes e essenciais em linguagem C.

Vamos, então, aprender a usá-lo.
A sintaxe é a seguinte:

```
while( condição )
{
   //código que
   //será repetido
}
```

Ou seja, o código do bloco só será rodado enquanto a condição der resultado verdadeiro.

Primeiro o while testa a condição, se for verdadeira, ele roda.
Ao término, ele testa novamente e roda novamente se for verdadeira.
Caso, em algum momento, o teste forneça valor lógico FALSO, o while vai parar e seu programa segue o fluxo normalmente.

Essas 'rodadas' de 'testa e roda' do while, e de outros laços, são chamadas de iterações.

E se a condição for sempre verdadeira? Como em:

```
while( 1 )
{
   printf("O que acontece?\n");
}
```

Acontece um looping infinito. Esse printf, ou qualquer código naquele lugar, rodaria indefinidamente...só pararia quando você fechasse o aplicativo ou reiniciasse sua máquina.

Na grande maioria dos códigos em C, porém, essa condição vai mudando. Geralmente é um número que cresce ou decresce, e ao atingir certo ponto, o while recebe FALSE como resultado do teste da condição, e o while termina.

Vamos ver alguns exemplos.

- **Exemplo comentado de código:**

Programa em C que conta de 1 até 10 usando o laço while

Primeiro declaramos uma variável, e inicializamos ela com o valor inicial, que é 1.
Logo em seguida ,fazemos com que o while dê um printf, mostrando o valor dessa variável ENQUANTO ela for MENOR ou IGUAL a 10.
Após imprimir o número 10, a variável é incrementada e passa a ter valor 11.
Como 11 não é menor ou igual a 10, o laço recebe valor lógico FALSE e termina sua execução.

```
#include <stdio.h>

int main(void)
{
	int numero=1;

	while(numero <= 10)
	{
	   printf("%d\n", numero);
	   numero++;
	}

}
```

Outros testes que resultariam positivo para esse exemplo:

```
while(numero < 19);
while(numero != 11);
```

Exemplo: Programa em C que conta de 10 até 1, usando o laço while
Agora vamos fazer o contrário do exemplo passado.
A lógico é a mesma: veja, raciocine e tente entender sozinho:

```
#include <stdio.h
int main(void)
{
	int numero=10;

	while(numero >= 1)
	{
	   printf("%d\n", numero);
	   numero--;
	}

}
```

- **Exemplo comentado de código:**

Crie um aplicativo em C que peça ao usuário o primeiro elemento de uma P.A, a razão da P.A e o número de elementos a serem exibidos.

O termo inicial P.A será armazenado na variável init.
A razão da P.A será armazenado na variável rate, e o número de elementos da P.A será armazenado na variável term.

Vamos usar duas variáveis de apoio, para contar.
Uma delas é a variável num, que representará o valor de cada termo da P.A. O valor dela será mudado a cada iteração, percorrendo todos os valores da P.A.
Já a variável count será usada para contar os termos da P.A. Os termos começam no primeiro (1) e terminar no último (term).

A fórmula para saber o valor do elemento 'an' de uma P.A é:
an = a1 + (n - 1) * razão

Assim, nosso código em C fica:

```
#include <stdio.h>

int main(void)
{
	int init,
	    rate,
	    term;

	int num,
	    count=1;

	printf("Digite o numero inicial da P.A: ");
	scanf("%d", &init);

	printf("Digite a razao da P.A: ");
	scanf("%d", &rate);

	printf("Digite o numero de termos da P.A: ");
	scanf("%d", &term);

	while(count <= term)
	{
	    num = init + (count - 1) * rate;
	    printf("Termo %d: %d\n",count, num);
	    count++;
	}

}
```

- **Exemplo comentado de código:**

Crie um aplicativo em C que peça ao usuário o primeiro elemento de uma P.G, a razão da P.G e o número de elementos a serem exibidos.
O raciocínio é análogo ao do exemplo anterior. Mas o termo 'an' de uma P.G é diferente da P.A, e é dado por:
an = a1 * (razao)^(n-1)

A única diferença aqui é que vamos usar a função *pow* da biblioteca *math.h.* O uso dela é bem simples: *pow*(*a,b)* significa 'a elevado a b'.

Então, nosso código em C fica:

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

int main(void)
{
	int init,
	    rate,
	    term;

	int num,
	    count=1;

	printf("Digite o numero inicial da P.G: ");
	scanf("%d", &init);

	printf("Digite a razao da P.G: ");
	scanf("%d", &rate);

	printf("Digite o numero de termos da P.G: ");
	scanf("%d", &term);

	while(count <= term)
	{
	    num = init*pow(rate,(count-1));
	    printf("Termo %d: %d\n",count, num);
	    count++;
	}
}
```

Questões sobre o laço WHILE em C

Na aula passada, em nosso curso online C Progressivo, aprendemos sobre o tão famoso e importante laço WHILE, da linguagem C.

Vamos agora propor algumas questões para você treinar e fixar seus conhecimentos.
Tente usar somente o laço while, e nenhum outro.

Não vá olhar a solução nem o código antes de tentar, bastante, resolver por conta própria. Do contrário, nunca aprenderá C.

- **Exercícios sobre o laço WHILE**

0. Programa em C dos patinhos da Xuxa
Xuxa, a rainha dos baixinhos, criou uma música que tem o segunte formato:

n patinhos foram passear
Além das montanhas
Para brincar
A mamãe gritou: Quá, quá, quá, quá
Mas só n-1 patinhos voltaram de lá.

Que se repete até nenhum patinho voltar de lá.
Ao final, todos os patinhos voltam:

A mamãe patinha foi procurar
Além das montanhas
Na beira do mar
A mamãe gritou: Quá, quá, quá, quá
E os n patinhos voltaram de lá.

Crie um programa em C que recebe um inteiro positivo do usuário e exibe a música inteira na tela, onde o inteiro recebido representa o número inicial n de patinhos que foram passear.

1. Programa em C que mostra os números ímpares
Escreva um aplicativo em C mostra todos os números ímpares de 1 até 100.

2. Programa em C que mostra os números pares
Escreva um aplicativo em C mostra todos os números pares de 1 até 100.

3. Programa em C que mostra os números pares e ímpares
Escreva um aplicativo em C que recebe inteiro e mostra os números pares e ímpares (separados), de 1 até esse inteiro.

4. Programa em C que calcula a média das notas de uma turma
Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.
Em seguida, através de um laço while, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.

Por fim, o programa mostra a média, aritmética, da turma.

5. Achando o maior número
Achar o maior, menor, média e organizar números ou sequências são os algoritmos mais importantes e estudados em Computação. Em C não poderia ser diferente.
Em nosso curso, obviamente, também não será diferente.

Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre qual destes números é o maior.

6. Achando os dois maiores números
Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre os dois maiores números digitados pelo usuário.

7. Quadrado de asteriscos
Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:
*****

*****
*****
*****
*****

8. Quadrado de asteriscos e espaços em branco
Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:
*****
* *
* *
* *
*****

9. Tabuada em C
Escreva um programa que pergunta um número ao usuário, e mostra sua tabuada completa (de 1 até 10).

10. Tabela ASCII em C
Se você lembrar bem, quando estudamos as variáveis do tipo caractere, *char*, dissemos que, na verdade, ela eram representadas por inteiros de 0 até 255.
Mostre a tabela completa do código ASCII.

Questões resolvidas sobre laço WHILE em C

0. Programa em C dos patinhos da Xuxa
Inicialmente, perguntamos ao usuário quantos patinhos ele quer que a mamãe pata tenha. Armazenaremos esse valor na variável inteira 'duck'. Vamos usar também outra variável inteira, a 'count', que recebe o mesmo valor da 'duck', mas será alterada dentro do laço while.

Após isso entramos em um while, que, enquanto o valor de 'count' for maior que 1, exibe o trecho inicial da música.
Nesse trecho inicial, basta substituirmos 'n' por 'count', e 'n-1' por 'count - 1'.

Quando count for 1, o laço while termina. Depois do while, a mamãe nota que nenhum patinho voltou, e depois vai buscar eles.

```
#include <stdio.h>

int main()
{
	int duck,
	    count;

	printf("Quantas patinhos a mamae Pata tem? ");
	scanf("%d", &duck);

	count = duck;
	while( count > 1)
	{
	    printf("%d patinhos foram passear\n", count);
	    printf("Além das montanhas\n");
	    printf("Para brincar\n");
	    printf("A mamãe gritou: Quá, quá, quá, quá\n");
	    printf("Mas só %d patinhos voltaram de lá.\n\n", count-1);

	    count--;
	}

	printf("1 patinho foi passear\n");
	printf("Além das montanhas\n");
	printf("Para brincar\n");
	printf("A mamãe gritou: Quá, quá, quá, quá\n");
	printf("Mas nenhum patinho voltou de lá.\n\n");
```

```
        printf("A mamãe patinha foi procurar\n");
        printf("Além das montanhas\n");
        printf("Na beira do mar\n");
        printf("A mamãe gritou: Quá, quá, quá, quá\n");
        printf("E os %d patinhos voltaram de lá.\n", duck);
}
```

1. Escreva um aplicativo em C mostra todos os números ímpares de 1 até 100.

Essa é bem simples. Basta percorrermos os números de 1 até 100, e fazer um teste condicional IF dentro do laço WHILE, para exibir somente aqueles números que são ímpares, ou seja, aqueles números que deixam resto 1 quando divididos por

```
#include <stdio.h>

int main()
{
        int count = 1;

        while(count <= 100)
        {
            if(count%2 == 1)
                printf("%d ",count);

            count++;
        }
}
```

2. Escreva um aplicativo em C mostra todos os números pares de 1 até 100.

Esse não precisa nem de código, basta você alterar apenas um número do exercício anterior.

3. Escreva um aplicativo em C que recebe inteiro e mostra os números pares e ímpares (separados, em duas colunas), de 1 até esse inteiro.

Como o primeiro número é ímpar (1), os ímpares serão exibidos primeiro. Após cada ímpar damos o espaço de um TAB (\t), e na mesma linha imprimimos o par, e logo em seguinte o caractere new line (\n).

```
#include <stdio.h>

int main()
```

```
{
	int num,
	    count = 1;

	printf("Digite um numero: ");
	scanf("%d", &num);

	printf("IMPARES \tPARES\n");

	while(count <= num)
	{
	    if(count%2 == 1)
	        printf("    %d \t",count);
	    else
	        printf("    \t %d\n", count);

	    count++;
	}
}
```

4.Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.
Em seguida, através de um laço while, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.
Por fim, o programa deve mostrar a média, aritmética, da turma.

Para resolver este exercício, vamos precisar de várias variáveis de apoio.
A variável 'total' vai receber o número total de alunos em uma sala.
A variável 'count' será a que vai mudar dentro do laço WHILE. E ela começa do primeiro aluno e termina no último.

A cada iteração do laço while, pedimos uma nota, que é somada a variável do tipo float 'soma'.

A média aritmética é dada por essa soma (de todas as notas), dividida pelo número de alunos na sala.
Assim, nosso programa em C fica:

```
#include <stdio.h>

int main()
```

```
{
	int total,
	    count = 1;
	float nota,
	      soma=0.0;

	printf("Numero de alunos na sala: ");
	scanf("%d",&total);

	while(count <= total)
	{
	    printf("Nota do aluno %d: ",count);
	    scanf("%f",&nota);

	    soma = soma + nota;

	    count++;
	}

	printf("Media da turma: %.2f", soma/total);
}
```

5. Achar o maior, menor, média e organizar números ou seqüências são os algoritmos mais importantes e estudados em Computação. Em C não poderia ser diferente.
Em nosso curso, obviamente, também não será diferente.

Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre qual destes números é o maior.

Vamos usar a nossa velha e conhecida variável 'count' pra contar de 1 até 10 no laço while.

Inicialmente, também, criamos a variável 'maior', que irá armazenar o maior número. Inicializamos ela com valor 0.

A cada iteração do laço while pedimos um número ao usuário, e armazenamos sempre na variável 'num'.
Em seguida fazemos uma comparação com o teste condicional IF ELSE para saber se esse número digitado é maior que o 'maior'. Se for, é porque esse número é o maior que já foi digitado, então fazemos com a variável 'maior' receba o valor desse 'num'.

Caso o 'num' não seja maior que 'maior', nada acontece e 'maior' continua armazenando o maior número.

Veja como ficou nossa solução em C:

```c
#include <stdio.h>

int main()
{
	int maior=0,
	    num,
	    count=1;

	while (count <= 10)
	{
	    printf("Digite o numero %d: ", count);
	    scanf("%d", &num);

	    if( num > maior)
	        maior=num;

	    count++;
	}

	printf("Maior: %d", maior);
}
```

6. Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final mostre os dois maiores números digitados pelo usuário.

Os dois primeiros números são armazenados fora do laço while.
Em seguida, é feita uma lógica simples entre esses dois para saber qual vai ser o maior e qual vai ser o segundo maior.

Agora que já temos os números certos nas variáveis 'maior' e 'segundo_maior', vamos ao laço WHILE que vai pedir do número 3 até o número 10.

Após o número ser inserido, ele vai ser armazenado na variável 'num'.
Agora vamos aos testes:
1. Primeiro checamos se 'num' é maior que 'maior', se for, é porque esse número será o novo número maior, e o antigo maior número temos que

colocar na variável 'segundo_maior'.
2. Caso não seja maior que 'maior', pode ser que ele seja maior que 'segundo_maior' e fazemos o teste.
Se for, somente o 'segundo_maior' muda de valor.

E se 'num' não for maior que 'maior' nem 'segundo_maior', nenhuma alteração é feita.

```
#include <stdio.h>

int main(main)
{
        int maior,
            segundo_maior,
            num,
            count=3;

        printf("Digite o numero 1: "); scanf("%d", &maior);
        printf("Digite o numero 2: "); scanf("%d", &num);

        if(maior > num)
            segundo_maior = num;
        else
        {
            segundo_maior = maior;
            maior = num;
        }


        while (count <= 10)
        {
            printf("Digite o numero %d: ", count);
            scanf("%d", &num);

            if(num > maior)
            {
                segundo_maior = maior;
                maior = num;
            }
            else
                if(num > segundo_maior)
                    segundo_maior = num;

            count++;
```

```
        }

        printf("Maior: %dn", maior);
        printf("Segundo maior: %dn", segundo_maior);

        return 0;
}
```

7. Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Por exemplo, para lado igual a 5:

****
****
****
****
****

Note que sempre existirão 'lado * lado' asteriscos. Vamos usar o while para imprimir todos os 'lado*lado' asteriscos.
Note também que a cada 'lado' asteriscos, ocorre uma quebra da linha.

No exemplo do lado=5: quando chegamos no asterisco de número 5, não imprimimos somente '*', mas '*\n'
O mesmo para o asterisco de número 10, pro de número 15, ...e pro de número 20
Ou seja, após o asterisco múltiplo de 5 ser impresso na tela, devemos colocar uma quebra da linha.

Como chamamos números que são múltiplos de 'lado'? Ora, pelo módulo (ou resto da divisão).

```
#include <stdio.h>

int main()
{
        int lado,
            count=1;
```

```
        printf("Lado do quadrado: ");
        scanf("%d", &lado);

        while( count <= lado*lado)
        {
            if(count % lado == 0)
                printf("*\n");
            else
                printf("*");

            count++;
        }

}
```

8. Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:

```
*****
*   *
*   *
*   *
*****
```

Primeiro vamos imprimir a primeira linha, que tem 'lado' asteriscos com um while.
Fazemos o mesmo para a última linha.

Agora vamos imprimir as linhas que não são completamente preenchidas de asteriscos, as que tem espaços em branco.
Note que essas linhas possuem um asterisco na primeira e na última posição.

Antes nós imprimíamos lado*lado caracteres. Agora, como tiramos duas linhas, vamos imprimir só 'lado*(lado-2)' linhas.

Se o caractere que formos imprimir for o primeiro da fila, imprimimos '*'.
Para sabermos isso, basta fazer um teste para ver se o contador (que vai de

1 até lado*(lado-2)) deixa resto 1 quando dividido por 'lado'.

Se for o último asterisco da linha, devemos imprimir '*\n', pois irá pular de linha.
Para detectar esse último asterisco, basta ver que é o caractere que, quando dividido por 'lado', deixa resto 0.

Se não for o primeiro nem o último caractere, imprimimos um espaço em branco: ' '

```
#include <stdio.h>

int main()
{
	int lado,
	    count=1;

	printf("Lado do quadrado: ");
	scanf("%d", &lado);

	//Imprimindo a primeira linha
	while(count<=lado)
	{
	    printf("*");
	    count++;
	}
	printf("\n");

	count=1;
	while( count <= lado*(lado-2))
	{
	    if( (count%lado == 1))
	        printf("*");
	    else
	        if( (count%lado == 0))
	            printf("*\n");
	        else
	            printf(" ");

	    count++;
	}

	//Imprimindo a primeira linha
```

```
        count=1;
        while(count<=lado)
        {
            printf("*");
            count++;
        }
        printf("\n");


}
```

9. Escreva um programa que pergunta um número ao usuário, e mostra sua tabuada completa (de 1 até 10).

Primeiro pedimos um número ao usuário, e fazemos com que esse número seja multiplicado por 1, por 2, ...até por 10.

Esse número que muda (1, 2, 3, ...,10), será gerado pelo laço while e uma variável de apoio 'count':

```
#include <stdio.h>

int main()
{
        int num,
            count=1;

        printf("Digite um numero: ");
        scanf("%d", &num);

        while(count <= 10)
        {
            printf("%d * %d = %d\n", num, count, num*count);
            count++;
        }
}
```

10. Se você lembrar bem, quando estudamos as variáveis do tipo caractere, char, dissemos que, na verdade, ela eram representadas por inteiros de 0 até 255.

Mostre a tabela completa do código ASCII.

Basta usarmos o laço while e uma variável de apoio para imprimirmos todos os números, de 0 até 255.

Mas ao invés de mostrar o número com o '%d', vamos usar '%c', pois assim iremos mostrar o caractere correspondente a cada número desses.

```
#include <stdio.h>

int main()
{
	int count=0;

	while(count <= 255)
	{
	    printf("%d = %c\n", count, count);
	    count++;
	}
}
```

O laço FOR: o que é, para que serve e como usar o FOR - Cast em C: o que é e como usar o Casting

O curso C Progressivo vai te ensinar agora o que é, provavelmente, o laço mais importante e usado na linguagem de programação C: o laço FOR, que será usado várias vezes durante nossa apostila.

- **O que é o laço FOR**

Se você notar bem em nossa aula sobre o laço WHILE, vai notar que precisamos inicializar uma variável - chamamos ela de 'count' - para usar dentro dos parênteses do WHILE, e incrementamos ou decrementamos essa variável ao fim do laço WHILE.

Com o tempo isso se torna um pouco incômodo, pois tem que inicializar e fazer operações em lugares diferentes de seu código.

Talvez agora você não veja muitos problemas e complicações, mas com o passar do tempo e suas aplicações em C forem ficando mais complexas, isso de inicializar e mudar o valor das variáveis se torna um pouco bagunçado.

No decorrer desse tutorial de C, e de outros artigos do site, você verá que existem muitas ferramentas que servem como atalho para os programadores.
Esses atalhos visam aumentar a eficiência do programador C, para que não perca muito tempo com detalhes.

Podemos ver o laço FOR como um desses atalhos, pois, em suma, o que é possível fazer com o laço WHILE, é possível fazer com o laço FOR.

Por que não usar o WHILE, então?
Porque o laço FOR é mais simples e eficiente de ser usado, na maioria das vezes.

Mas não há nada que nos impeça de usar o laço WHILE, e algumas vezes ele até mais útil que o FOR.

- **A sintaxe do laço FOR: como usar o for**

A sintaxe é a seguinte:

```
for(inicio_do_laço ; condição ; termino_de_cada_iteração)
{
//código a ser
//executado aqui
}
```

Isso quer dizer que, ao iniciar o laço for, ele faz o que está no trecho "inicio_do_laço".
Geralmente se usa para inicializar algumas variáveis (o que fazíamos antes de iniciar o laço WHILE).

Após inicializar, o for testa a condição.
Se ela resultar verdadeira, o código é executado e em seguida o 'termino_de_cada_iteração" é executado.
A condição é testada novamente, e se for verdadeira, executa o 'termino_de_cada_iteração" também, e assim continua até que a condição resulte no valor lógico FALSO (0) e o laço FOR termina.

Vamos ver na prática como usar o laço FOR através de exemplos de programas em C.

- **Exemplo: Contando de 1 até 10 com o laço FOR**

No exemplo a seguir, apenas declaramos a variável 'count', e iniciamos ela dentro do laço for.
O primeiro teste é se '1 <= 10', como é verdadeiro, o código que imprime o número é executado.
Após o término da execução, o count é incrementado em 1. Agora count=2.

Na nova iteração, testamos a seguinte condição '2 <= 10', que resulta em valor lógico VERDADEIRO, por tanto o código é executado e o número 2 impresso. Agora incrementamos a variável 'count', que agora vale 3.

E assim por diante, até imprimirmos o valor 10 e 'count' ser incrementada e se tornando count=11.
No próximo teste, a condição agora é FALSA e o laço for termina.

```
#include <stdio.h>

int main(void)
{
	int count;

	for(count=1 ; count <= 10 ; count++)
	{
	   printf("%d\n", count);
	}
}
```

- **Exemplo:**

Contagem regressiva de 10 até 1, usando o laço FOR

O raciocínio é igual ao do exemplo anterior, porém vamos decrementar a variável que vai se iniciar com 10.
Veja como ficou nosso código em C:

```
#include <stdio.h>

int main(void)
{
	int count;

	for(count=10 ; count >= 1 ; count--)
	   printf("%d\n", count);
}
```

Assim como no teste condicional IF ELSE, e no laço WHILE, se o código que vem após o laço tem apenas uma linha, não é necessário usar as chaves.
Se você colocar duas, ou mais, linhas de código após o IF, ELSE, WHILE ou FOR, apenas a primeira linha fará parte do teste/laço.

Muita atenção pra isso! Se não quiser correr riscos, você pode usar sem problema algum o par de chaves.

- **Exemplo: Contagem progressiva e regressiva no mesmo laço FOR**

Ora, se no laço WHILE podemos usar, inicializar, comparar, testar e operar matematicamente várias variáveis, também podemos fazer isso no laço FOR.

Agora inicializamos duas variáveis, testamos se ambas obedecem a uma determinada condição, e usamos dois operadores matemáticos, tudo ao mesmo tempo. Veja esse código em C e tente entender como funciona:

```
#include <stdio.h>

int main(void)
{
	int up,
	    down;

	printf("CRESCENTE \tDECRESCENTE\n");
	for(up=1, down=10 ; up<=10 && down >=1 ; up++, down--)
	{
	    printf("    %d \t\t    %d\n", up, down);
	}
}
```

- **Exemplo:**

Crie um programa em C que gera os elementos de uma P.A pedindo ao usuário o número de elementos da P.A, sua razão e seu elemento inicial.

Lembrando que a fórmula do n-ésimo termo da P.A é: an = a1 + (n-1)*razao
Fica fácil ver que:

```
#include <stdio.h>

int main(void)
{
```

```
    int termos,
        razao,
        inicial,
        count;

    printf("Número de termos da P.A: ");
    scanf("%d", &termos);

    printf("Razão da P.A: ");
    scanf("%d", &razao);

    printf("Elemento inicial da P.A: ");
    scanf("%d", &inicial);

    for(count = 1; count <= termos ; count++)
        printf("Termo %d: %d\n", count, (inicial + (count-1)*razao) );

}
```

- **Exemplo:**

Crie um programa em C que gera os elementos de uma P.G pedindo ao usuário o número de elementos da P.G, sua razão e seu elemento inicial.

O exemplo dessa questão é igual ao que demos artigo sobre o laço WHILE. Porém, vamos apresentar agora o casting.

A função pow, da biblioteca math.h recebe dois números decimais e retorna um decimal também, do tipo double.
Porém, não queremos trabalhar com o tipo double, e sim com o tipo inteiro, então temos que dizer e sinalizar isso pra função pow, de alguma maneira.

Essa maneira é o cast, que é colocar o tipo de variável que queremos obter, antes do retorno da função ou antes de uma variável.
No nosso caso, queremos que a função pow() retorne um inteiro, então fizemos: (int) pow(...)

Ou seja, basta colocarmos o tipo que queremos entre parênteses, e teremos esse tipo de variável.

Mas só fizemos isso porque o resultado de nossa P.G é inteiro. Caso usássemos decimais e colocássemos o cast pra inteiro, iríamos ficar só com a parte inteira do número decimal e perderíamos os valores decimais.

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

int main(void)
{
	int termos,
	    razao,
	    inicial,
	    elemento,
	    count;

	printf("Número de termos da P.G: ");
	scanf("%d", &termos);

	printf("Razão da P.G: ");
	scanf("%d", &razao);

	printf("Elemento inicial da P.G: ");
	scanf("%d", &inicial);

	for(count = 1; count <= termos ; count++)
	    printf("Termo %d: %d\n", count, inicial* (int)pow(razao,count-1) );
}
```

Questões sobre o laço FOR

Como já havíamos explicado no artigo sobre o laço FOR em C, esse laço é uma maneira mais simples e eficiente de fazer coisas que eram possíveis de se fazer com o laço WHILE.

O curso C Progresseivo vai propor agora as mesmas questões que já havíamos colocado para o laço WHILE, mas agora você deve tentar resolver apenas usando o laço FOR.

- **Exercícios sobre o laço FOR**

0. Programa em C dos patinhos da Xuxa
Xuxa, a rainha dos baixinhos, criou uma música que tem o segunte formato:

n patinhos foram passear
Além das montanhas
Para brincar
A mamãe gritou: Quá, quá, quá, quá
Mas só n-1 patinhos voltaram de lá.

Que se repete até nenhum patinho voltar de lá.
Ao final, todos os patinhos voltam:

A mamãe patinha foi procurar
Além das montanhas
Na beira do mar
A mamãe gritou: Quá, quá, quá, quá
E os n patinhos voltaram de lá.

Crie um programa em C que recebe um inteiro positivo do usuário e exibe a música inteira na tela, onde o inteiro recebido representa o número inicial n de patinhos que foram passear.

1. Programa em C que mostra os números ímpares
Escreva um aplicativo em C mostra todos os números ímpares de 1 até 100.

2. Programa em C que mostra os números pares
Escreva um aplicativo em C mostra todos os números pares de 1 até 100.

3. Programa em C que mostra os números pares e ímpares
Escreva um aplicativo em C que recebe inteiro e mostra os números pares e ímpares (separados), de 1 até esse inteiro.

4. Programa em C que calcula a média das notas de uma turma
Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.
Em seguida, através de um laço while, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.

Por fim, o programa mostra a média, aritmética, da turma.

5. Achando o maior número
Achar o maior, menor, média e organizar números ou sequências são os algoritmos mais importantes e estudados em Computação. Em C não poderia ser diferente.
Em nosso curso, obviamente, também não será diferente.

Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre qual destes números é o maior.

6. Achando os dois maiores números
Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre os dois maiores números digitados pelo usuário.

7. Quadrado de asteriscos
Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:
*****
*****
*****
*****
*****

8. Quadrado de asteriscos e espaços em branco
Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:
*****
* *
* *

```
*   *
*****
```

9. Tabuada em C
Escreva um programa que pergunta um número ao usuário, e mostra sua tabuada completa (de 1 até 10).

10. Tabela ASCII em C
Se você lembrar bem, quando estudamos as variáveis do tipo caractere, *char*, dissemos que, na verdade, ela eram representadas por inteiros de 0 até 255.
Mostre a tabela completa do código ASCII.

Soluções das questões sobre o laço FOR

0. Programa em C dos patinhos da Xuxa

```c
#include <stdio.h>

int main()
{
        int duck,
            count;

        printf("Quantos patinhos a mamae Pata tem? ");
        scanf("%d", &duck);

        for(count=duck ; count!=1; count--)
        {
            printf("%d patinhos foram passear\n", count);
            printf("Além das montanhas\n");
            printf("Para brincar\n");
            printf("A mamãe gritou: Quá, quá, quá, quá\n");
            printf("Mas só %d patinhos voltaram de lá.\n\n", count-1);
        }

        printf("1 patinho foi passear\n");
        printf("Além das montanhas\n");
        printf("Para brincar\n");
        printf("A mamãe gritou: Quá, quá, quá, quá\n");
        printf("Mas nenhum patinho voltou de lá.\n\n");

        printf("A mamãe patinha foi procurar\n");
        printf("Além das montanhas\n");
```

```
        printf("Na beira do mar\n");
        printf("A mamãe gritou: Quá, quá, quá, quá\n");
        printf("E os %d patinhos voltaram de lá.\n", duck);
}
```

1. Programa em C que mostra os números ímpares

```
#include <stdio.h>

int main()
{
        int count;

        for(count=1 ; count<= 100 ; count++)
        {
            if(count%2 == 1)
                printf("%d ",count);

            count++;
        }
}
```

2. Programa em C que mostra os números pares

3. Escreva um aplicativo em C que recebe inteiro e mostra os números pares e ímpares (separados, em duas colunas), de 1 até esse inteiro.

```
#include <stdio.h>

int main()
{
        int num,
            count;

        printf("Digite um numero: ");
        scanf("%d", &num);

        printf("IMPARES \tPARES\n");

        for(count=1 ; count<=num ; count++)
        {
            if(count%2 == 1)
                printf("    %d \t",count);
            else
                printf("    \t %d\n", count);
        }
```

```
}
```

4. Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.
Em seguida, através de um laço while, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.
Por fim, o programa deve mostrar a média, aritmética, da turma.

```
#include <stdio.h>

int main()
{
	int total,
	    count;
	float nota,
	      soma=0.0;

	printf("Numero de alunos na sala: ");
	scanf("%d",&total);

	for(count=1, soma ; count<=total ; count++)
	{
	    printf("Nota do aluno %d: ",count);
	    scanf("%f",&nota);

	    soma = soma + nota;
	}

	printf("Media da turma: %.2f", soma/total);
}
```

5. Achar o maior, menor, média e organizar números ou seqüências são os algoritmos mais importantes e estudados em Computação. Em C não poderia ser diferente.
Em nosso curso, obviamente, também não será diferente.

Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final
mostre qual destes números é o maior.

```c
#include <stdio.h>

int main()
{
	int maior,
	    num,
	    count;

	for(count=1, maior=0; count<=10 ; count++)
	{
	    printf("Digite o numero %d: ", count);
	    scanf("%d", &num);

	    if( num > maior)
	        maior=num;
	}

	printf("Maior: %d", maior);
}
```

6. Escreva um programa em C que solicita 10 números ao usuário, através de um laço while, e ao final mostre os dois maiores números digitados pelo usuário.

```c
#include <stdio.h>

int main()
{
	int maior,
	    segundo_maior,
	    num,
	    count;

 printf("Digite o numero 1: ");
	scanf("%d", &maior);

	printf("Digite o numero 2: ");
	scanf("%d", &segundo_maior);

 if(maior < segundo_maior)
 {
    count = maior;
    maior = segundo_maior;
    segundo_maior = count;
 }
```

```
        for(count=3; count<=10 ; count++)
        {
            printf("Digite o numero %d: ", count);
            scanf("%d", &num);

            if(num > maior)
            {
                segundo_maior = maior;
                maior = num;
            }
            else
  if(num > segundo_maior)
   segundo_maior = num;
        }

        printf("Maior: %d\n", maior);
        printf("Segundo maior: %d\n", segundo_maior);
}
```

7. Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Por exemplo, para lado igual a 5:

```
*****
*****
*****
*****
*****
```

```
#include <stdio.h>

int main()
{
        int lado,
            count;

        printf("Lado do quadrado: ");
        scanf("%d", &lado);

        for(count=1 ; count<=lado*lado ; count++)
        {
            if(count % lado == 0)
```

```
            printf("*\n");
        else
            printf("*");
    }


}
```

8. Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.
Para lado igual a 5:

```
*****
*   *
*   *
*   *
*****
```

```
#include <stdio.h>

int main()
{
    int lado,
        count;

    printf("Lado do quadrado: ");
    scanf("%d", &lado);

    //Imprimindo a primeira linha
    for(count=1 ; count<=lado; count++)
        printf("*");

    printf("\n");

    for(count=1 ; count<=lado*(lado-2) ; count++)
    {
        if( (count%lado == 1))
            printf("*");
        else
            if( (count%lado == 0))
                printf("*\n");
            else
                printf(" ");
```

```
        }

        //Imprimindo a primeira linha
        for(count=1 ; count<=lado; count++)
            printf("*");

        printf("\n");


}
```

9. Escreva um programa que pergunta um número ao usuário, e mostra sua tabuada completa (de 1 até 10).

```
#include <stdio.h>

int main()
{
        int num,
            count;

        printf("Digite um numero: ");
        scanf("%d", &num);

        for(count=1 ; count<=10 ; count++)
            printf("%d * %d = %d\n", num, count, num*count);
}
```

10. Se você lembrar bem, quando estudamos as variáveis do tipo caractere, char, dissemos que, na verdade, ela eram representadas por inteiros de 0 até 255.
Mostre a tabela completa do código ASCII.

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

        for(count=0 ; count<=255 ; count++)
            printf("%d = %c\n", count, count);
}
```

Os comandos CONTINUE e BREAK: pausando e alterando o fluxo de laços e testes condicionais

Até o momento, em nosso curso de C, sempre percorremos os laços do início até o fim, no intervalo que escolhemos.

Porém, nem sempre isso é necessário. Pode ser que no meio do caminho consigamos obter o resultado que esperávamos e a continuação do laço seria mera perda de tempo.

Para controlar melhor os laços, vamos apresentar os comandos BREAK e CONTINUE, que usaremos ao longo de toda nossa apostila de C.

- **O comando BREAK em C: como usar**

Break, em inglês, significa pausa, ruptura ou parar bruscamente.

E é basicamente isso que esse comando faz com os laços: se você colocar o comando break dentro de um laço e este for executado, ele vai parar imediatamente o laço e o programa continua após o laço.

Vamos dar um exemplo prático, e você verá melhor como funciona e qual a utilidade do comando break em C, que também existe em diversas outras linguagens de programação, como C++ e Java.

Sua principal função é evitar cálculos desnecessários.
Muitas vezes rodamos um looping, através de um laço (como FOR ou WHILE), para encontrar alguma informação. O break serve para terminar o laço em qualquer momento (como no momento em que encontrarmos essa informação, ao invés de ter que esperar que o looping termine).

- **Exemplo de código comentando em C**

Ok, assumimos.
A explicação que demos pode parecer um pouco confusa, a primeira vista.

Mas como sempre, você vai entender melhor o uso dos comandos BREAK e CONTINUE através de exemplos resolvidos e comentados de questões.

Vamos mostrar alguns exemplos, e você entenderá melhor para que serve e como usar os comandos BREAK e CONTINUE, em C.

Ache o primeiro número, entre 1 e 1 milhão que é divisível por 11, 13 e 17.
Isso quer dizer que temos que encontrar o menor número que, quando dividimos ele por 11, por 13 e por 17 dê resto da divisão nulo.

Uma possível solução é a seguinte:

```
#include <stdio.h>

int main()
{
	int count,
	    numero=0;

	for(count=1 ; count<=1000000 ; count++)
	{
	    if(numero == 0)
	        if((count%11==0) && (count%13==0) && (count%17==0))
	            numero=count;
	}

	printf("O numero e: %d", numero);

}
```

A variável 'count' é a que irá contar de 1 até 1 milhão.
O número que queremos será armazenado em 'numero' e inicializamos com valor 0.
Enquanto 'numero' tiver valor 0, é porque o número que estamos procurando ainda não foi achado, pois quando ele for achado a variável 'numero' irá mudar de valor e passará a ser o número que queremos.

Note que esse valor ficará sempre armazenado na variável 'numero', pois agora o primeiro teste condicional nunca mais será satisfeito.

Você vai ver que o número em questão é 2431.
E o que acontece depois que descobrimos esse número?

Ora, o laço continua a contar de 2431 até 1000000, pra nada.
Então o computador está gastando processamento à toa.

Aqui na minha máquina, essa operação durou 10milisegundos
Vamos dar uma otimizada nisso usando o comando BREAK.

- **Exemplo de código comentado em C**

Ache o primeiro número, entre 1 e 1 milhão que é divisível por 11, 13 e 17. Use o comando BREAK.

A idéia do break é simples: quando acharmos o valor que queremos, damos um break e o programa sai do laço.
O código vai ficar assim:

```
#include <stdio.h>

int main()
{
	int count,
	    numero;

	for(count=1 ; count<=1000000 ; count++)
	    if((count%11==0) && (count%13==0) && (count%17==0))
	    {
	        numero=count;
	        break;
	    }


	printf("O numero e: %d", numero);

}
```

Note que agora só precisamos de um teste condicional IF.
Aqui na minha máquina, essa operação levou 2milisegundos.
Ou seja, 5x mais rápida. Isso se deve ao fato do laço não precisar mais ir de 2432 até 1 milhão.
Quando ele achou o número que eu queria, ele parou de contar.

Você pode pensar: "Ah, 8milisegundos não fazem a mínima diferença".

Bom, pra esse tipo de aplicação simples, realmente não faz mesmo, é praticamente instantâneo, ainda mais na linguagem C que é reconhecidamente uma das mais rápidas.

Agora imagine um jogo de alta performance ou o código de um Sistema Operacional, onde alguns códigos e rotinas são rodados milhões ou até bilhões de vezes? Esses 5ms virariam segundos ou minutos de lag ou travamento.
Por isso é sempre bom fazer um código limpo e eficiente, sem perder processamento à toa.

O comando break será mais utilizado no próximo artigo do C Progressivo, sobre o teste condicional SWITCH.

- **O comando CONTINUE em C: como usar**

O comando CONTINUE, quando inserido dentro de algum laço, faz com que a iteração atual seja cancelada, e o laço prossegue na próxima iteração.

Ou seja, o BREAK faz todo o laço parar.
Já o CONTINUE, faz somente com que a iteração atual pare, pulando pra próxima.

Exemplo:
Faça um aplicativo em C que some todos os números, de 1 até 100, exceto os múltiplos de 5.

Fazemos o laço for percorrer de 1 até 100.
Testamos se cada número deixa resto 0 quando dividido por 5. Caso deixe, é porque é múltiplo de 5 e não devemos somar.

Para não somar, simplesmente pulamos essa iteração.

Porém, se não for múltiplo de 5, é porque a iteração não foi pulada, ela continua, então vamos somar esse número na soma total.

```
#include <stdio.h>

int main()
{
```

```
    int count,
        soma;

    for(count=1, soma=0 ; count<=100 ; count++)
    {
        if( count%5 ==0)
            continue;

        soma = soma + count;
    }


    printf("Soma %d", soma);

}
```

Diferente do BREAK, que pode ser usado em laços e no SWITCH (que veremos a seguir), o CONTINUE é utilizado em laços somente.

O teste condicional SWITCH

Provavelmente você nunca notou mas, a todo instante, estamos fazendo escolhas quando usamos um computador ou outro dispositivo digital qualquer.

Escolhemos ícones, teclas do teclado, que botão vamos clicar, que menus vamos abrir, que opção do menu vamos selecionar etc.

O curso C Progressivo vai ensinar, agora, uma ferramenta importantíssima em programação: o uso do switch para escolhas.

- **SWITCH em C: o que é e como usar o comando**

SWITCH é um comando em C que serve para fazer testes condicionais, testando igualdades, onde podemos usar várias opções de comparações.

Assim como o nosso conhecido par IF ELSE, o comando switch também serve para fazer testes condicionais.

Imagine que você quer testar um valor digitado pelo usuário com 10 números.
Você poderia fazer com IF, tranquilamento.
Porém, seu código iria ficar enorme e você teria que digitar várias vezes IF (...), IF(...)

Visando reduzir isso, vamos aprender como usar o comando *switch*, que iremos usar várias vezes durante nossa apostila de C, para criar menus, por exemplo, onde iremos exibir uma série de opções, o usuário vai escolher uma e vamos saber qual opção ele escolheu através de um comando *switch*.

A sintaxe do switch é a seguinte:

```
switch( opção )
{
   case opção1:
         comandos caso a opção 1 tenha sido escolhida
         break;

   case opção2:
```

```
        comandos caso a opção 2 tenha sido escolhida
        break;

    case opção3:
        comandos caso a opção 3 tenha sido escolhida
        break;

    default:
        comandos caso nenhuma das opções anteriores tenha sido escolhida
}
```

O switch vai comparar a variável 'opção' com os 'case'. Se ele achar uma opção (case) que seja igual, ele vai rodar o código que vem após esse case, e antes do próximo case.
Caso nenhum case seja igual a 'opção', o código que está default é o que será rodado.

Caso a 'opção' seja um char, coloque entre aspas simples ' ', caso seja string coloque entre aspas duplas " " e caso seja um número, não é necessário colocar nenhum tipo de aspas.

- **Exemplo de código:**

Crie uma calculadora usando a instrução SWITCH, que pergunte qual das operações básicas quer fazer (+, -, * e /), em seguida peça os dois números e mostre o resultado da operação matemática entre eles.

```
#include <stdio.h>

int main()
{
    char operacao;
    float num1,
        num2;

    printf("Escolha sua operação [+ - * / ]: ");
    scanf("%c",&operacao);

    printf("Entre com o primeiro número: ");
    scanf("%f",&num1);
```

```
        printf("Entre com o segundo número: ");
        scanf("%f",&num2);

        switch( operacao )
        {
            case '+':
                printf("%.2f + %.2f = %.2f", num1, num2, num1 + num2);
                break;

            case '-':
                printf("%.2f - %.2f = %.2f", num1, num2, num1 - num2);
                break;

            case '*':
                printf("%.2f * %.2f = %.2f", num1, num2, num1 * num2);
                break;

            case '/':
                printf("%.2f / %.2f = %.2f", num1, num2, num1 / num2);
                break;

            default:
                printf("Você digitou uma operacao invalida.");

        }

}
```

- **O switch sem o break**

No exemplo passado, você viu que somente um dos case era selecionado para cada operação que efetuamos.

Porém, isso só ocorre por conta do break em cada case.

Se você tirar o break, o switch identifica o case que é igual a 'operacao', executa ele e TODOS OS QUE ESTÃO ABAIXO até o fim, ou até encontrar um break!

É como se os case se acumulassem.

Rode o exemplo a seguir:

```c
#include <stdio.h>

int main(void)
{
	int continuar=1;
	char opcao;

	while(continuar)
	{
	   system("clear || cls");
	   printf("Repetir? (S/s)im (N/n)ao\n");
	   scanf(" %c",&opcao);

	   switch(opcao)
	   {
	      case 's':
	      case 'S':
	         break;

	      case 'n':
	      case 'N':
	         continuar = 0;
	   }

	}

}
```

O comando system("clear") serve para limpar a tela em sistemas operacionais do tipo Linux, e system("cls") limpa a tela caso você use Windows. Então system("clear || cls") limpa a tela, independente de qual sistema você esteja usando.

Nota-se que, enquanto continuar=1, o laço WHILE continua a ocorrer e só termina quando 'continuar' receber valor 0.

Se digitarmos 's' o primeiro case é selecionado. Como ele não tem break, o próximo também ocorre, que é o case caso opcao='S'.
Esse tem break. Ou seja, pra continuar a repetir basta digitar 'S' ou 's'.

Se digitar 'n', vai cair no case onde opcao='n' e onde opcao='N', pois não tem break no opcao='n'.

Então, 'continuar' recebe valor 0 e o laço WHILE termina.

- **Exemplo:**

Suponha que você atrasou uma conta. A cada mês que você deixa de pagar, será cobrado 1% de juros no valor inicial.
Ou seja, se você atrasar um mês, irá pagar 1%. Se atrasar 3 meses, irá pagar 3% etc.
Vamos supor que você pode atrasar, no máximo, 5 meses.

O programa pede, como entrada, dois valores:
- um float: com o valor de sua dívida inicial (valor_i)
- um inteiro: de 0 até 5, que são os meses de atraso.

Faça um programa em C que calculo o juros de atraso. Use switch e case acumulados.

```
#include <stdio.h>

int main(void)
{
	float valor_i,
	      valor_f;
	int   juros=0;

	int meses;

	printf("Qual o valor inicial da dívida: ");
	scanf("%f", &valor_i);

	printf("Você vai atrasar quantos meses [1-5]?: ");
	scanf("%d", &meses);

	switch( meses )
	{
	   case 5:
	      juros++;
	   case 4:
	      juros++;
	   case 3:
	      juros++;
```

```
            case 2:
                juros++;
            case 1:
                juros++;
                break;
            default:
                printf("Você não digitou um valor válido de meses\n");

        }
        printf("Juros: %d%\n",juros);
        valor_f=( (1 + (juros/100.0))*valor_i);
        printf("Valor final da dívida: R$ %.2f\n", valor_f);

}
```

Caso tenha atrasado 5 meses, o valor da variável 'juros' é incrementado 5 vezes.
Se atrasou 4 meses, o 'juros' é incrementado 4 vezes, e assim por diante.

Exercício:
Crie um programa que receba um inteiro, de 1 até 12, representando os meses do ano e retorne o número de dias do mês.
Use switch e não use break. Acumule os case.

Inicialmente, a variável 'dias' é declarada como tendo 31 dias.

Caso seja o mês 4, 6, 9 ou 11, é subtraído 1 da variável 'dias' e o programa informa que esses meses possuem 30 dias.

Caso seja o mês 2, é subtraído 2 de 'dias', ficando 28 dias para o mês de fevereiro.

Caso não seja nenhum desses meses, não cai no switch, então continua com 31 dias (que são os meses 1, 3, 5, 7, 8, 10 e 12).

Assim, nosso código C fica:

```c
#include <stdio.h>

int main()
{
        int mes,
            dias=31;
```

```c
        printf("Digite o mes [1-12]: ");
        scanf("%d", &mes);

        if(mes>12 || mes<1)
            printf("Mes invalido");
        else
            switch( mes )
                {
                    // fevereiro: subtraímos 2 dias aqui e 1 dia no próximo case
                    case 2:
                        dias -=2;

                    //meses que possuem 30 dias: só subtraímos 1 dia
                    case 4: case 6: case 9: case 11:
                        dias--;

                }

        printf("O mes %d possui %d dias", mes, dias);

}
```

O que é o laço DO WHILE

Vamos apresentar o último, e não menos importante, laço de nosso curso C Progressivo: o laço do ... while.

Como o nome pode sugerir, ele tem muito a ver com o laço WHILE, que já estudamos e treinamos seu uso em tutoriais passados de nossa **apostila de C**.

- **Para que serve o laço DO WHILE em C**

Se você notar bem, em nosso artigo sobre o laço WHILE, pode reparar que esse laço só roda se a condição for verdadeira.
Se essa condição for uma variável ou comparação, por exemplo, devemos sempre nos certificar que, inicialmente, essa condição vai ser verdadeira.

Geralmente inicializamos as variáveis ou pedimos algum dado pro usuário, tudo para nos certificarmos que o programa vai entrar no WHILE, o que as vezes pode ser um pouco incômodo e nada eficiente.

O que o laço DO WHILE faz é executar, PELO MENOS UMA VEZ, o que está dentro dele e só ao final da execução é que ele faz o teste, usando o velho e conhecido laço WHILE.

Ou seja, temos a garantia que o laço vai ser executado uma vez, sempre precisar inicializar variável ou pedir dados ao usuário antes do WHILE (pois fazer isso pode bagunçar o sistema).

- **Como declarar e usar o laço DO WHILE em C**

Lembrando que DO, em inglês e nesse contexto, significa "faça" e WHILE significa "enquanto".

Ou seja, esse laço quer dizer "faça isso" -> código -> "enquanto essa condição for verdadeira, repita" (note como os nomes dos comandos não são escolhidos de forma aleatória, tudo em programação C faz sentido e foi criado com propósitos de facilidade a vida do programador e aumentar sua eficiência).

Resumindo, ele sempre executa seu código ao menos uma vez, o que é uma mão na roda em certas ocasiões.

E qual a diferença do laço DO WHILE para o WHILE ?
Bom, para o WHILE rodar é necessário que a condição seja sempre verdadeira.
Logo, se ela for inicialmente FALSE, o looping while nunca irá rodar.

Já o DO WHILE vai rodar sempre, ao menos uma vez, mesmo que a condição existente no WHILE seja falsa (se for falsa, vai rodar só uma vez e termina).

A sintaxe desse comando é a seguinte

```
do
{
   //código que
   //será repetido
   //aqui

} while (condição);
```

Vale realçar aqui que esse laço, diferente dos outros, **POSSUI PONTO E VÍRGULA** no final! Não se esqueçam!

No mais, não há segredos, é igual ao WHILE, a lógica é a mesma. Sabendo bem o WHILE, saberá bem o laço DO WHILE.

Vamos mostrar um exemplo em que o DO WHILE é bem usado em programação C: exibição de menu.

- **Exemplo de uso do laço DO WHILE: Criando um MENU**

Crie um menu que apresente algumas opções ao usuário, usando o laço DO WHILE e o teste condicional SWITCH em C, para selecionar as opções.
Uma dessas condições é o término da exibição do menu.

Vamos usar apenas uma variável, o inteiro 'opcao' que irá selecionar a opção que o usuário quiser.
Dentro do escopo do do {}, mostramos o menu de opções e pedimos pro usuário digitar algo.

A condição do while é a própria variável.
Se o usuário digitar 0, o WHILE irá receber FALSE como condição e irá encerrar.
Caso o WHILE receba qualquer outra coisa diferente de 0, receberá valor lógico TRUE e irá continuar o laço DO WHILE.

Pois bem, já sabemos o que ocorre quando 0 é digitado.
Então, dentro do SWITCH, quando a opção 0 é selecionada, apenas limpamos a tela e damos um aviso de saída.

Caso 1 ou 2 seja digitado, idem: limpamos a tela e mostramos uma mensagem.

Caso 3 ou 4 seja escolhido, limpamos a tela e não acontece nada, simplesmente o menu é exibido novamente.

Se o usuário não digitar nenhuma dessas opções, é porque ele digitou algo errado e vai cair no case DEFAULT do SWITCH, e informamos que ele errou na escolha da opção.

Assim, nosso código de menu fica:

```
#include <stdio.h>

int main(void)
{
	int opcao;

	do
	{
	   printf("\t\t\nMenu do curso C Progressivo\n");
	   printf("0. Sair\n");
	   printf("1. Dar Boas vindas\n");
	   printf("2. Dar Oi\n");
	   printf("3. Repetir o menu\n");
	   printf("4. Ler mais uma vez o menu\n");

	   printf("Opcao: ");
	   scanf("%d", &opcao);

	   switch( opcao )
```

```
        {
            case 0:
                system("cls || clear");
                printf("Saindo do menu...\n");
                break;
            case 1:
                system("cls || clear");
                printf(" Bem-vindo ao menu do portal C Progressivo! \n");
                break;
            case 2:
                system("cls || clear");
                printf(" Oi! \n");
                break;
            case 3:
            case 4:
                system("cls || clear");
                break;
            default:
                system("cls || clear");
                printf("Opcao invalida! Tente novamente.\n");
        }
    } while(opcao);

}
```

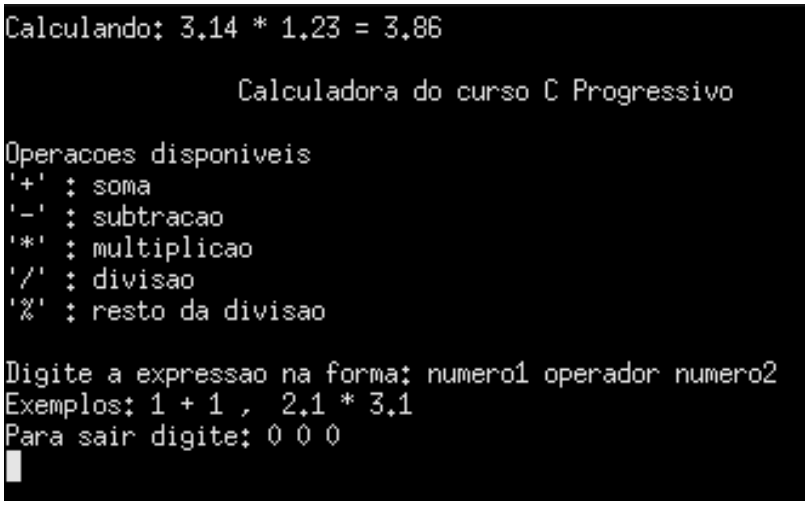
Obviamente esse é um exemplo simples, que não faz nada, mas mostra bem o funcionamento do DO WHILE.
E menus são, obviamente, muito usados em programação.

Em nossa apostila online de C, vamos usar menus em aplicativos de simulação de um sistema bancário, em jogos e vários outros aplicativos. Portanto, é essencial que tenha entendio bem o funcionamento do DO WHILE com SWITCH.

- **Exercício de C:**

No artigo sobre o teste condicional SWITCH em C, mostramos como usar ele para fazer uma calculadora.
Refaça essa calculadora, com o mesmo SWITCH, mas agora mostrando as operações matemáticas como opções de um menu, dentro de um laço DO WHILE.

Criando uma calculadora em C



No artigo passado, sobre o laço DO WHILE em C, propomos para você um exercício:

No artigo sobre o teste condicional SWITCH em C, mostramos como usar ele para fazer uma calculadora.
Refaça essa calculadora, com o mesmo SWITCH, mas agora mostrando as operações matemáticas como opções de um menu, dentro de um laço DO WHILE.

Vamos resolver esse exercício e comentar totalmente seu código, agora em nossa **apostila de C**!

- ## Como criar uma calculadora em C

Você já saiu do básico e aprendeu todos os testes condicionais e laços na linguagem.
Já está na hora de fazer algo realmente útil, que você possa se orgulhar e até mostrar aos amigos.

Ou seja, vamos fazer uma calculadora que fazer as operações de soma, subtração, multiplicação, divisão e resto da divisão.

Tudo isso com base nos conhecimentos que acumulamos até aqui.

Obviamente, não será uma calculadora complexa e de utilidade geral, é mais para termos noção de como usar os conhecimentos que aprendemos até aqui.

Ela também não é uma aplicação 'robusta', que seja blindada e segura. Ela é facilmente 'hackeável'.
Por exemplo, ela está programada para receber números, então se você digitar uma letra, irá 'quebrar' nosso programa.

Mas ao passo que vamos estudando e evoluindo em nossa apostila, iremos criar aplicações cada vez melhores e mais seguras
Pois bem, vamos começar!

- **A lógica de uma calculadora na linguagem C**

Vamos usar apenas três variáveis nesse aplicativo C: dois floats (que vão representar os dois números) e uma variável do tipo char, que irá armazenar o tipo de operação o usuário quer: +, -, *, / ou %

Conforme pedimos, o menu será exibido através do laço DO WHILE. Lembrando que temos que colocar uma condição para esse laço continuar. Você pode criar a sua, a minha foi a seguinte: se o usuário digitar num1=0, oper=0 e num2=0, o programa termina.

Pois bem, mostramos o menu ao usuário (meros *printf*, que você já domina totalmente) e demos exemplos de como deve ser a entrada do usuário. Que deve ser: **numero operador numero**

Se colocarmos os 3 *scanf* em sequência, dá para pegar o primeiro número, o char e em seguida o segundo número, mas o usuário tem que entrar com os dados da seguinte maneira: digita o primeiro número, dá um espaço, digita o operador e dá outro espaço.

Após isso, limpamos a tela e mostramos ao usuário a operação matemática que ele escolheu:
numero operador numero2 =

E o resultado dessa operação? Hora, vai depender do operador e números que ele tenha escolhido.

Essa resposta vai ser fornecida através do switch.

Mandamos o operador para o switch, que vai selecionar a operação e mostrar um printf com o resultado.

Caso o usuário tenha digitado um operador inválido, o switch manda isso pro default.
Lá no default temos que fazer uma espécie de tratamento de informação, para saber se o usuário digitou um operador inválido ou se digitou 0 0 0, pois se tiver digitado 0 0 0 é porque ele quer terminar o programa.

Isso é facilmente resolvido com o nosso conhecido e amado teste condicional IF ELSE.

```
#include <stdio.h>

int main(void)
{
	float num1,
	      num2;
	char oper;

	do
	{
	   printf("\t\tCalculadora do curso C Progressivo\n\n");

	   printf("Operacoes disponiveis\n");
	   printf("'+' : soma\n");
	   printf("'-' : subtracao\n");
	   printf("'*' : multiplicao\n");
	   printf("'/' : divisao\n");
	   printf("'%%' : resto da divisao\n");

	   printf("\nDigite a expressao na forma: numero1 operador numero2\
n");
	   printf("Exemplos: 1 + 1 ,  2.1 * 3.1\n");
	   printf("Para sair digite: 0 0 0\n");


	   scanf("%f", &num1);
	   scanf(" %c",&oper);
	   scanf("%f", &num2);
```

```
        system("cls || clear");

        printf("Calculando: %.2f %c %.2f = ", num1,oper,num2);


        switch( oper )
        {
           case '+':
                 printf("%.2f\n\n", num1 + num2);
                 break;

           case '-':
                 printf("%.2f\n\n", num1 - num2);
                 break;

           case '*':
                 printf("%.2f\n\n", num1 * num2);
                 break;

           case '/':
                 if(num2 != 0)
                    printf("%.2f\n\n", num1 / num2);
                 else
                    printf("Nao existe divisao por 0\n\n");
                 break;

           case '%':
                 printf("%d\n\n", (int)num1 % (int)num2);
                 break;

           default:
                 if(num1 != 0 && oper != '0' && num2 != 0)
                    printf(" Operador invalido\n\n ");
                 else
                    printf(" Fechando calculadora!\n ");
        }

     }while(num1 != 0 && oper != '0' && num2 != 0);

}
```

Exercícios sobre testes condicionais e laços

Para fechar mais uma seção com chave de ouro, vamos propor algumas questões para você fazer, mas não vamos dizer "faça usando isso", "faça daquele jeito" ou "aplique esse conhecimento".

Afinal, na vida real como programador C, ninguém vai te dizer como você deve fazer seu trabalho.
Você tem que usar sua criatividade, tem que pensar e desenrolar.

Essas questões, portanto, não tem apenas uma única e correta solução. Cada programador vai ter ideias diferentes e maneiras diferentes de resolver o problema.

Pense, relembre as aulas e tente resolver da maneira mais simples, eficiente e bonita...ou simplesmente tente resolver, já será um passo importante.

Use:
Teste condicional IF ELSE
Laço WHILE
Laço FOR
Teste condicional SWITCH, com ou sem CONTINUE e BREAK
Laço DO WHILE

Fique a vontade para criar sua própria solução, da maneira que mais lhe convier.

0. Escreva um programa em C que recebe 'n' números do usuário, e recebe o número 'n' também, e determine qual destes números é o menor.

1. Escreva um programa em C que recebe um inteiro 'n' do usuário e calcula o produto dos números pares e o produtos dos números ímpares, de 1 até n.

2. Faça um programa em C que recebe um inteiro do usuário e calcula seu fatorial.
O fatorial de 'n' é dado por:
n*(n-1)*(n-2)...*3*2*1
e é representado por n!

3. Crie um aplicativo bancário em C que pede o valor do depósito inicial, o valor do investimento mensal e o número de meses que o dinheiro vai ficar rendendo na poupança.
Após isso, calcule o lucro obtido, sabendo que o juros da poupança é de 0,5%.

4. Crie um programa em C que peça um número inteiro ao usuário, e imprima a seguinte tabela::
1
2 4
3 6 9
4 8 12 16
...
5. Escreva um programa que peça um númer 'n' ao usuário, e que gere um novo **n** de acordo com a seguinte regra:

• se n é par, n = n / 2
• se n é impar, n = 3 * n + 1
• imprime n
• O programa deve parar quando x tiver o valor □igual a 1. Por exemplo, para n = 13, a saída será:
40 -> 20 -> 10 -> 5 -> 16 -> □8 -> 4 -> 2 -> 1

Desafio dos números de FIbonacci
Crie um aplicativo em C que peça um número inteiro ao usuário - **'n'** - e exiba o n-ésimo termo da série de Fibonacci, sabendo que o primeiro termo é 0, o segundo é 1 e o próximo número é sempre a soma dos dois anteriores.

Desafio do diamante de asteriscos:
Escreva um aplicativo em C que peça um número inteiro ímpar ao usuário e desenhe um diamante no seguinte formato:

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Nesse caso, o número é 9, pois há 9 colunas e 9 asteriscos na linha central.

Super hiper mega desafio de Fibonacci
Fazer o desafio dos números de FIbonacci, mostrador anteriormente, usando apenas duas variáveis.

Soluções

Questão 0:
Vamos usar quatro variáveis na solução desse exercício: o 'n' vai armazenar quantos números o usuário vai inserir, o 'count' será usado como contador do laço, o 'num' é a variável que vai armazenar o número que o usuário digitar a cada iteração e por fim temos a variável 'menor' que, como o nome diz, vai armazenar o valor do menor número inserido.

A lógica da questão é simples: a cada laço, pedimos um número ao usuário. Se esse número for o primeiro a ser inserido, ele será o menor até então.

Depois disso, continuamos a pedir os números ao usuário, e comparamos se esse número inserido é menor que o valor do número armazenado em 'menor', se for menor, a variável 'menor' vai receber esse novo valor.

Fazendo isso em todas as iterações, terminamos o laço com o menor valor inserido dentro da variável 'menor'.

```
#include<stdio.h>

int main()
{
    int n,
        count,
        num,
        menor;

    printf("Quantos numeros deseja comparar: ");
    scanf("%d", &n);

    for(count = 1 ; count <= n ; count++)
    {
        printf("Insira o numero %d: ", count);
        scanf("%d", &num);

        if(count == 1)
```

```
            menor=num;

        if(num < menor)
            menor=num;
    }

    printf("Menor numero: %d", menor);
}
```

Questão 1:
Como de praxe, a variável 'n' recebe quantos números serão exibidos (de 1 até n) e a variável 'count' será a responsável pelo controle de nosso laço.

Iniciamos as variáveis 'pares' e 'impares', que irão receber o produto, com valor 1 (consegue pensar o motivo disso?)

A seguir entramos no laço, e dentro deste fazemos um teste condicional do tipo IF ELSE.
Testamos se o número é par. Se for, ele deixará resto da divisão por 2 igual a 0, então multiplicamos esse número por 'pares'.
Caso o número seja ímpar, vamos multiplicar esse 'count' pela variável 'impares'

Assim, ao final do laço, teremos o produto de todos os pares e de todos os números ímpares de 1 até n, nas variáveis 'pares' e 'impares', respectivamente.

Relembrando que os seguintes símbolos:
pares *= count;
impares *= count;

Representam o mesmo que:
pares = pares * count;
impares = impares * count;

Assim, o código de nossa questão será:

```
#include<stdio.h>

int main()
{
    int n,
        count,
        pares=1,
```

```
        impares=1;

    printf("Produto dos pares e impares ate o numero: ");
    scanf("%d", &n);


    for(count = 1 ; count <= n ; count++)
    {
        if(count % 2 == 0)
            pares *= count;
        else
            impares *= count;
    }

    printf("\nProduto dos pares: %d\n", pares);
    printf("\nProduto dos impares: %d\n", impares);
}
```

Questão 2:
Essa questão é uma continuação da anterior. Se fez a questão 1, fará essa com facilidade.
Porém essa tem uma utilidade muito grande: calcular fatoriais é muito importante em Matemática.
Assim, criando esse programa você terá criado uma ferramenta muito útil.

Pois bem, a variável 'n' e a 'count' você já sabe o que faz.
Agora, para armazenar o produto de 1 até n, vamos usar só a variável 'res', que nos fornecerá a resposta final.

O laço não tem segredo, basta fazer rodar de 1 até n, e a cada iteração multiplicamos esse número da iteração por 'res', que ao final temos o número desejado, o fatorial de n: n!

```
#include<stdio.h>

int main()
{
    int n,
        count,
        res=1;

    printf("Insira um inteiro positivo: ");
    scanf("%d", &n);
```

```
    for(count = 1 ; count <= n ; count++)
        res *= count;

    printf("\nFatorial de %d eh: %d\n", n, res);
}
```

Questão 3:
O funcionamento é o seguinte:
Você inicia sua conta com um valor 'inicial'.
Depois de 1 mês, rendeu 0,5%, ou seja, você teve um lucro de 0.005 * inicial
Agora seu total é: inicial + 0,005*inicial = 1,005 inicial

Essa fórmula será usada sempre: o dinheiro que você tem no início do mês, é 1,005 daquele que você tinha há um mês atrás.
Então, 'valor_final' = '1,005*valor_final'

Após esse lucro gerado pela poupança, você vai adicionar, todo mês, um 'investimento'.
Assim, após o lucro e investimento, seu dinheiro é: valor_final = 1,005*valor_final + investimento

Logo, nosso código C fica:

```
#include<stdio.h>

int main()
{
    float inicial = 0.0,
          investimento = 0.0,
          valor_final;

    int meses=0,
        count;

    printf("Investimento inicial: ");
    scanf("%f", &inicial);
    valor_final = inicial;

    printf("Investimento mensal: ");
    scanf("%f", &investimento);

    printf("Tempo, em meses, que deixara rendendo: ");
    scanf("%d", &meses);
```

```
    for(count = 1 ; count <= meses ; count++)
    {
        valor_final = 1.005*valor_final;
        valor_final += investimento;
        printf("Apos %d mes(es) voce tera: %.2f\n", count, valor_final);
    }

    printf("\nVoce comecou com %.2f e investiu %.2f durante %d meses\n",
inicial, investimento, meses);
    printf("Seu lucro foi de %.2f\n", valor_final - inicial - (meses*investimento));
}
```

Questão 4:
Note que o usuário vai fornecer o número de linha e que o número da linha, é o primeiro dígito da linha.
Por exemplo, a linha 3 começa com o número 3.

Note também que estamos calculando os múltiplos.
Por exemplo, a linha 'n' tem elementos que são o 'n' multiplicado de 1 até n: n*1, n*2, n*3, ...,n*n

Vamos usar dois laços for, encadeados.
O primeiro, é o que vai ficar responsável pela linha. Essas linha vão de 1 até n.

Dentro de cada linha dessa, vamos imprimir 'n' elementos. Ou seja, vamos imprimir as colunas.
Cada elemento da coluna é o valor do número da linha multiplicado pelo número da coluna (as colunas variam de 1 até n).

Então, nosso código fica assim:

```
#include<stdio.h>

int main()
{
    int num,
        linha,
        coluna;

    printf("Entre com o numero: ");
    scanf("%d", &num);

    for(linha = 1 ; linha <= num ; linha++)
    {
```

```
        for(coluna = 1 ; coluna <= linha ; coluna++)
            printf("%3d ", linha*coluna);

        printf("\n");
    }
}
```

Questão 5:
Uma maneira simples de fazer essa questão é com o laço while, pois as operações que deverão ser feitas, devem acontecer ENQUANTO o número for diferente de 1.

Dentro desse laço, usamos o resto da divisão do número por 2, pra saber se o número é par ou ímpar e então a operação é

```
#include <stdio.h>

int main()
{
        int num;

        printf("Digite um numero inteiro: ");
        scanf("%d", &num);

        while(num != 1)
        {
            if( num%2 == 0)
                num = num/2;
            else
                num = 3*num +1;

            printf("%d ", num);
        }
}
```

Desafio dos números de FIbonacci:
Vamos chamar o n-ésimo termo da série de 'numero', o (n-1)-ésimo termo de 'ultimo' e o (n-2)-ésimo de 'penultimo'.
Assim, temos a fórmula geral: numero = ultimo + penultimo;

Para calcularmos o próximo termo da série ('numero'), temos que mudar os valores de 'ultimo' e 'penultimo'.
Para tal, o valor de 'penultimo' agora será o valor de 'ultimo', e o valor de

'ultimo' será agora o valor de 'numero'. Após fazer isso, podemos calcular o novo valor de 'numero', que é 'ultimo + penultimo'.

Por exemplo, chegamos em um ponto que:
numero = 21
ultimo = 13
penultimo = 8

Para calcular o próximo termo, fazemos 'penultimo=ultimo', ou seja, agora:
penultimo = 13
Analogamente, fazemos 'ultimo=numero', e temos:
ultimo=21

De posse desses novos valores, temos o novo valor de 'numero', que é: 21 + 13 = 34, que é o próximo termo da série.

```
#include<stdio.h>

int main()
{
    int numero,
        ultimo=1,
        penultimo=0,
        count,
        termo;

    printf("Entre com o termo desejado: ");
    scanf("%d", &termo);

    for( count = 3 ; count <= termo ; count++)
    {
        numero = ultimo + penultimo;
        penultimo=ultimo;
        ultimo=numero;
    }

    printf("Elemento de termo %d: %d\n", termo, numero);
}
```

Desafio do diamante de asteriscos:
Há duas coisas na figura: espaços em branco e asteriscos.
Vamos resolver esse desafio analisando o padrão do número de asteriscos (variável asteriscos) e do número de espaços em branco (variável espacos).

Antes de desenhar o diamante, temos que checar se o número que o usuário

forneceu é ímpar, através de um teste condicional:
if(numero%2 != 0)

Se não for, o programa cai no else e nossa aplicação C é finalizada.
Vamos desenhar o dito cujo.

Parte de cima do diamante
Qual o número inicial de espaços?
Note que é sempre: (numero-1)/2
...e essa é a lógica do problema, a parte mais difícil, que é notar esse padrão.

E o número de asteriscos inicial?
Fácil, é 1 e vai crescendo de 2 em 2 até que o número de asteriscos impressos seja igual a numero.

Pois bem, vamos desenhar linha por linha, imprimindo primeiro o número correto de espaços e depois o de asteriscos.
Esse controle de linhas é feito pela linha de código:
for(linha=1 ; espacos > 0 ; linha++)

Ou seja, vamos imprimir da linha 1 até a linha central (note que a linha central é a que tem espacos=0).

Agora, dentro desse looping vamos imprimir, em cada linha, primeiro o número de espaços e depois os asteriscos.
Sabemos que o número inicial de espaços é (numero-1)/2 e o número inicial de asteriscos é 1.

Para imprimir o número correto de espaços e asteriscos vamos usar a variável count:
for(count=1 ; count <= espacos ; count++)
for(count=1 ; count <= asteriscos ; count++)

Após o looping maior, o das linhas, temos que decrementar o número de espaços em 1, e incrementar o número de asteriscos em 2, além da quebra de linha.

Parte de cima do diamante
A lógica é a mesma da de cima, porém o número de espaços aumenta em 2, e o número de asteriscos é decrementado em 2.
Outra diferença é que vamos imprimir uma linha a menos, pois a linha central já foi impressa. Assim, essas linhas são impressas desde a primeira abaixo

da linha central, até enquanto houver asteriscos (até ter 1, que é o último):
for(linha=1 ; asteriscos > 0 ; linha++)

```c
#include<stdio.h>

int main()
{
	int numero,
	    espacos,
	    asteriscos,
	    count,
	    linha;

	printf("Insira um numero impar: ");
	scanf("%d", &numero);

	if(numero%2 != 0)
	{
	    //Imprimindo a parte de cima do diamante
	    asteriscos = 1;
	    espacos = (numero-1)/2;
	    for(linha = 1 ; espacos > 0 ; linha++)
	    {
	        //Espaços
	        for(count = 1 ; count <= espacos ; count++)
	            printf(" ");

	        //Asteriscos
	        for(count = 1 ; count <= asteriscos ; count++)
	            printf("*");

	        espacos--;
	        asteriscos += 2;
	        printf("\n");
	    }

	    //Imprimindo a parte de baixo do diamante
	    for(linha=1 ; asteriscos > 0 ; linha++)
	    {

	        //Espaços
	        for(count = 1 ; count <= espacos ; count++)
	            printf(" ");

	        //Asteriscos
```

```
            for(count = 1 ; count <= asteriscos ; count++)
                printf("*");

            espacos++;
            asteriscos -= 2;
            printf("\n");
        }

    }
    else
        printf("Não é ímpar!");

}
```

Super hiper mega desafio de Fibonacci
Essa questão é um pouco mais complicada, vai precisar de conhecimentos em Programação e um pouco de Matemática, mas não deixa de ser simples e engenhosa.

Na seção passada usamos três variáveis: 'numero', 'ultimo' e 'penultimo', para descrever o n-ésimo, (n-1)-ésimo e o (n-2)-ésimo termo, respectivamente.
Vamos excluir a variável 'numero', e trabalhar somente com o 'ultimo' e 'penultimo'.

Vamos supor que em um determinado ponto temos:
ultimo=5
penultimo=3

É fácil ver que o próximo termo é sempre (ultimo + penultimo), que nesse caso é 8.
Nosso objetivo então é fazer com que as variáveis tenham o valor ultimo=8 e penultimo=5, concorda?

Para que consigamos gerar mais números de Fibonacci, fazemos com que 'ultimo' receba seu antigo valor somado com o valor de 'penultimo', pois assim a variável 'ultimo' terá a soma dos dois números anteriores, como diz a regra de Fibonacci. Em C, fica assim:
ultimo = ultimo + penultimo

Agora ultimo = 8, e penultimo = 3.
Nosso objetivo agora é fazer com que penultimo receba o valor 5. Mas como, se nenhuma das variáveis possui valor 5? De onde obter esse valor?
De uma relação matemática da série de Fibonacci! Note que sempre: ultimo = numero - penultimo
Veja como obter 5: 5 = 8 -3 , ou seja, é o novo valor de 'ultimo' subtraído do atual valor de 'penultimo'.

Em C fica assim: penultimo = ultimo - penultimo

Note no desenho como variam os valores das variáveis no decorrer da execução:

Veja como fica o resultado:

```
#include<stdio.h>

int main()
{
    int ultimo=1,
        penultimo=0,
        termo,
        count;

    printf("Entre com o termo desejado: ");
    scanf("%d", &termo);

    printf("%d\n%d\n", penultimo, ultimo);

    for( count = 3 ; count <= termo ; count++)
    {
        ultimo = ultimo + penultimo;
        penultimo = ultimo - penultimo;
    }
    printf("Elemento de termo %d: %d\n", termo, ultimo);
}
```

Funções em C

Para acompanhar estes tutoriais, é necessário que você tenha estudado nossa seção anterior, de nosso **Curso de C**, sobre Testes e Laços.

Nessa seção de nosso site, vamos nos dedicar exclusivamente ao ensino das funções, uma parte importantíssima da linguagem de programação C.

Usaremos, de maneira natural e corriqueira, funções em praticamente todas nosss aplicações em C, de tão importante que é o conceito de funções em C.

Portanto, para seguir em nosso curso, é de vital importância que você entenda bem o que são funções, como e porque usamos elas.

O que são funções, para que servem e como usá-las em C

Para ter uma seção só para este assunto em nossa apostila online C Progressivo, você deve suspeitar que essas tais funções sejam bem importantes em C. E são mesmo, e muito.

Porém, se você lembrar bem, você já usou as benditas funções: *printf, scanf, system* e própria *main* é uma função. A partir de agora, vamos ensinar a você os conceitos de funções em C.

- **O que é uma função em linguagem C**

Função é um bloco de código que, como o próprio nome diz, tem uma função, uma tarefa específica.
As funções servem como um atalho à um bloco de código.
Veja função como uma maneira de se evitar a repetição de código.

Em vez de programar um trecho de código várias vezes, definimos uma função que faz tal tarefa e depois, ao longo do programa, basta invocar essa função que, automaticamente, o código nela é executado.

Desta maneira, podemos criar um trecho de código, que representa uma lógica, um passo ou rotina, do processo inteiro e isolá-lo.

Com a criação das funções, vamos ver como elas irão facilitar a vida de um programador, pois será necessário criar a função somente uma vez, e ela estará disponível e pronta para ser usada no futuro.

- **Para que serve uma função em C?**

Um dos motivos nós já expomos no tópico anterior, que é o fato de você economizar linhas de código.
O que é possível fazer com função, é possível fazer sem, usando os laços e testes condicionais estudados na unidade anterior.

Porém, nossos programas em C ficariam enormes, e essa falta de organização deixa inviável a criação de programas mais complexos. Se estudou todos os tutoriais de nosso curso, deve ter notado que alguns

programas ficaram bem extensos, e é difícil ter controle sobre centenas ou milhares de linhas de código.
Uma das utilidades das funções é, portanto, a organização e a redução de nossos códigos.

Quando um programador profissional inicia um projeto, ele já tem que ter boa parte do código pronto.
“Como ter código pronto, se você nem sabe que projeto vai participar nem o que o programa vai fazer?”

Todos os programadores, inclusive você que já é programador C, deve fazer a reusabilidade de código, e uma das maneiras mais comuns de se fazer isso é através de funções.

Durante seus anos de estudo da linguagem C, você irá criar centenas ou milhares de funções.
Então, atente bem para essa dica: **guarde e organize suas funções**.

Ainda não entendeu? Vamos dar um exemplo.
Vamos supor que uma empresa te contratou para fazer um sistema em C. Nesse sistema, serão necessários fazer cálculos matemáticos dos mais diversos.

Ora, se você já tiver feito uma função que soma números, outra que multiplica e por ai vai, você não vai precisar escrever elas de novo. Basta você ir em seu banco de dados de programação e copiar o código já feito. Isso é reusabilidade. Ou seja, você vai usar bastante os códigos que você programou durante toda sua vida.

Antes de iniciar nossos estudos sobre funções, vamos dar uma dica importantíssima, que frequentemente é ignorada, mesmo por programadores experientes: **suas funções devem fazer uma única coisa e muito bem feita**.

Por exemplo, em vez de fazer uma função que calcula as raízes de uma equação do segundo grau, faça uma função que calcula do delta (Bháskara) e outra que calcula a raiz quadrada de um número. Só então faça a função que calcula as raízes, usando as outras funções.

Não faça uma função que cria uma casa. Faça uma função que cria quartos, outra que cria banheiros, outra que cria sala, outra que cria um jardim, outra que cria um cozinha etc.

Por fim, faça uma função que invoca as funções anteriores, montando assim, sua casa.

Se um dia você precisar de uma função que constrói banheiro, você já tem ela separada. Afinal, banheiro é banheiro. Mas se tivesse feito uma função que criava a casa de uma vez, você não teria esse código, que cria o banheiro, de maneira acessível.

- **Como declarar funções em C**

A sintaxe de uma função é a seguinte

```
tipo_de_retorno nome_da_função(lista de parâmetros)
{
        // código da
          // função
}
```

E para invocar a função basta usar o comando: nome_da_função(lista de argumentos).

Quando aos parâmetros, argumentos e tipos de retorno, serão explicados em breve.
Por hora vamos usar funções que não retornam nada (funções do tipo *void*) e que não recebem nenhum argumento.

- **Exemplo de código:**

Função que mostra uma mensagem na tela.

Criamos a função hello(), que não retorna nada, por isso colocamos o void.
A função dessa função é simplesmente mostrar uma mensagem na tela, quando é invocada.

```c
#include<stdio.h>

void hello()
{
	printf("Ola mundo! Estou aprendendo C com o curso C Progressivo\n");
}

int main()
{
	hello();
}
```

Podemos colocar esse chamado da função em qualquer lugar da main, ou de outra função.

Poderíamos colocar dentro de um laço, e repetir o chamado dessa função quantas vezes quiséssemos. E a vantagem é que não precisamos escrever de novo o printf, basta chamar a função hello() que ela irá imprimir, automaticamente, a frase na tela.

Criando um menu usando funções em C

Para ilustrar bem o uso das funções, vamos repetir um exercício que fizemos quando estudamos testes condicionais e laços.

Aqui, vamos criar um aplicativo em C que exibe algumas frases e invoca funções.

- **Exemplo de código em C**

Crie um programa em C que exiba um menu e converse com o usuário.

Vamos exibir um menu, com algumas saudações, para o usuário escolher. Essa opção ficará armazenada na variável inteira 'continuar', que também será usada como condição para que o laço DO WHILE ocorra.

Assim, caso o usuário digite 0, ele vai sair do programa.
Caso ele digite qualquer outro número, a opção é tratada pelo teste condicional SWITCH.

Bem, se você estudou todas as aulas de nosso curso, sobre testes e laços, você já é um mestre.
Então não precisamos entrar muito em detalhes, apenas olhando para o código você já nota o que está acontece.

Nosso foco aqui é nas funções.
Note o SWITCH e seus *cases*, como ficaram menor, mais organizados e bonitos.

```
#include<stdio.h>

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

void tudo_bem()
{
   printf("Tudo otimo, e com voce?\n");
}

void familia()
```

```
{
    printf("Meus bits...digo, minha familia vai bem!\n");
}
void sair()
{
    printf("Ja vai??? Nao! Nao! Espere! Naa...\n");
}

int main()
{
    int continuar=1;
    do
    {
        printf("\n\tChat Foreve Alone\n\n");
        printf("1. Oi\n");
        printf("2. Tudo bem\n");
        printf("3. Como vai a familia\n");
        printf("0. Sair\n");

        scanf("%d", &continuar);
        system("cls || clear");

        switch(continuar)
        {
            case 1:
                oi();
                break;

            case 2:
                tudo_bem();
                break;

            case 3:
                familia();
                break;

            case 0:
                sair();
                break;

            default:
                printf("Digite uma opcao valida\n");
        }
    } while(continuar);
}
```

Argumentos e Parâmetros de funções em C - Funções aninhadas

Na aula passada, de introdução as funções em C, nós mostramos alguns exemplos de funções que exibiam uma mensagem na tela.

Mas é claro que as funções não são estáticas assim, e não fazem só isso. Através de parâmetros e argumentos, podemos enviar dados para as funções trabalharem e fazerem algo mais útil para nossos programas.

É o que faremos nesse tutorial de nossa apostila.

- **O que são e como usar parâmetros e argumentos em C**

Parâmetros são variáveis que a função recebe. O número e tipo de parâmetros são pré estabelecidos na declaração da função.
E essa declaração, do(s) tipo(s) e nome(s) da(s) variável(eis), acontece dentro dos parêntesis.

Por exemplo, se formos criar uma função que recebe e soma dois números, declaramos a função com os tipos de variáveis (inteiro ou float, por exemplo) e o nome dessas variáveis:

```
void soma(int a, int b)
{
}
```

Pronto, declaramos uma função com dois parâmetros.
Agora, dentro dessa função, podemos usar 'a' e 'b'. É como se elas tivessem sido declaradas entre parêntesis.

E quais os valores desses parâmetros?
São os que você vai passar. Esses valores são os argumentos.

Por exemplo, se você invocar a função: soma(1,2)
Os argumentos são 1 e 2, onde os parâmetros 'a' e 'b' vão receber os valores 1 e 2, respectivamente.
Ou seja, o valor que você colocar primeiro, vai pra primeira variável, e o segundo valor dentro dos parêntesis será o segundo parâmetros, e assim vai.

Note agora que essas funções não são estáticas, eles mudam de acordo com os valores que você passa pra ela. No exemplo a seguir, vamos mostrar como usar melhor as funções com parâmetros e argumentos.

Agora, antes de ir para o próximo exemplo de código, recomendamos que você dê um salto para o futuro e leia nossa aula sobre O que é e como usar o comando return em C.

Vamos aguardar você voltar.

- ### Exemplo de código

Em nosso tutorial sobre o laço DOWHILE, propomos e resolvemos um exercício, que criava uma calculadora usando o laço DO WHILE e o SWITCH.
Refaça essa calculadora. Agora, use funções para calcular as operações matemáticas.

A parte com testes condicionais IF ELSE e SWITCH, bem como a parte do laço DO WHILE, estão do mesmo jeito, a lógica permanece.

A diferença agora é que usamos funções, todas de parâmetros 'a' e 'b', e que recebem como argumentos os números que usuário digitar, que no caso são 'num1' e 'num2'.

Assim, em vez de realizar as operações dentro do SWITCH, deixamos que cada função faça essa tarefa. Isso tem um impacto muito grande na organização. Só em ler o nome da função você já sabe o que ela faz. Veja como ficou nosso aplicativo em C:

```
#include <stdio.h>

void soma(float a, float b)
{
   printf("%.2f\n",a+b);
}

void subtracao(float a, float b)
{
   printf("%.2f\n",a-b);
}
```

```c
void multiplicacao(float a, float b)
{
    printf("%.2f\n",a*b);
}

void divisao(float a, float b)
{
    if( b != 0)
        printf("%.2f\n",a/b);
    else
        printf("Nao pode dividor por zero\n");
}

void modulo(int a, int b)
{
    printf("%d\n", a%b);
}

void calculadora(int a, int b, char oper)
{

}


int main()
{
        float num1,
              num2;
        char oper;

        do
        {
            printf("\t\tCalculadora do curso C Progressivo\n\n");

            printf("Operacoes disponiveis\n");
            printf("'+' : soma\n");
            printf("'-' : subtracao\n");
            printf("'*' : multiplicao\n");
            printf("'/' : divisao\n");
            printf("'%%' : resto da divisao\n");

            printf("\nDigite a expressao na forma: numero1 operador numero2\
n");
            printf("Exemplos: 1 + 1 ,  2.1 * 3.1\n");
```

```
        printf("Para sair digite: 0 0 0\n");


        scanf("%f", &num1);
        scanf(" %c",&oper);
        scanf("%f", &num2);

        system("cls || clear");

        printf("Calculando: %.2f %c %.2f = ", num1,oper,num2);


        switch( oper )
        {
           case '+':
                soma(num1, num2);
                break;

           case '-':
                subtracao(num1, num2);
                break;

           case '*':
                multiplicacao(num1, num2);
                break;

           case '/':
                divisao(num1, num2);
                break;

           case '%':
                modulo(num1, num2);
                break;

           default:
                if(num1 != 0 && oper != '0' && num2 != 0)
                   printf(" Operador invalido\n\n ");
                else
                   printf(" Fechando calculadora!\n ");
        }

    }while(num1 != 0 && oper != '0' && num2 != 0);

}
```

- **Funções aninhadas - Invocando uma função dentro de outra função**

Vamos agora invocar uma função a partir de outra.
Na verdade sempre fizemos isso, pois a *main*() é uma função também.
O exemplo passado, da calculadora, será usado novamente.

Note que a lógica da questão é através da escolha dos números, operador e a execução do cálculo.

Como já fizemos as funções, que fazem os cálculos, separadamente, vamos fazer uma função que seleciona essa operação. Ou seja, vamos criar a função 'calculadora()', que recebe dois números e um char (que representa a operação) e já devolve a expressão e o valor.

Por questões de organização, vamos criar uma função chamada 'menu()', que simplesmente exibe o menu.

Agora, nosso DO WHILE chama menu(), depois recebe os números e operador, e passa essas variáveis para função *calculadora()*. E essa função é responsável por descobrir que operador recebeu e que operação vai fazer, chamando as outras funções.

Porém, vale ressaltar aqui um importante detalhe: para uma função ser chamada dentro de outra, ela já deve ter sido declarada ANTES dessa função em que ela está sendo chamada.

Ou seja: pra função *calculadora()* funcionar, as outras funções matemáticas devem ter sido declaradas anteriormente! O C não reconhece se você chamar uma função, e ela ainda não tiver sido declarada. Ele olha a ordem das declarações.

Veja como nosso programa em C ficou após usarmos as funções aninhadas:

```
#include <stdio.h>

void menu()
{
   printf("\t\tCalculadora do curso C Progressivo\n\n");
```

```
    printf("Operacoes disponiveis\n");
    printf("'+' : soma\n");
    printf("'-' : subtracao\n");
    printf("'*' : multiplicao\n");
    printf("'/' : divisao\n");
    printf("'%' : resto da divisao\n");

    printf("\nDigite a expressao na forma: numero1 operador numero2\n");
    printf("Exemplos: 1 + 1 , 2.1 * 3.1\n");
    printf("Para sair digite: 0 0 0\n");
}


void soma(float a, float b)
{
    printf("%.2f\n",a+b);
}

void subtracao(float a, float b)
{
    printf("%.2f\n",a-b);
}

void multiplicacao(float a, float b)
{
    printf("%.2f\n",a*b);
}

void divisao(float a, float b)
{
    if( b != 0)
        printf("%.2f\n",a/b);
    else
        printf("Nao pode dividor por zero\n");
}

void modulo(int a, int b)
{
    printf("%d\n", a%b);
}

void calculadora(float a, float b, char operador)
{
    system("cls || clear");
```

```
    printf("Calculando: %.2f %c %.2f = ", a,operador,b);

    switch( operador )
    {
    case '+':
        soma(a, b);
        break;

    case '-':
        subtracao(a, b);
        break;

    case '*':
        multiplicacao(a, b);
        break;

    case '/':
        divisao(a, b);
        break;

    case '%':
        modulo((int)a, (int)b);
        break;

    default:
        if(a != 0 && operador != '0' && b != 0)
            printf(" Operador invalido\n\n ");
        else
            printf(" Fechando calculadora!\n ");
    }
}

int main()
{
    float num1,
    num2;
    char oper;

    do
    {
        menu();

        scanf("%f", &num1);
        scanf(" %c",&oper);
        scanf("%f", &num2);
```

```
        calculadora(num1, num2, oper);
    }
    while(num1 != 0 && oper != '0' && num2 != 0);

}
```

Esse último exemplo é o mais próximo de um programa real em C, pois foram usadas funções com tarefas bem específicas e divididas.

Outro ponto importante é o fato da main() estar bem "enxuta".

Somente nos programas inicias, quando estamos aprendendo a linguagem C, é que a main() fica grande, com muito código. O ideal é usar ela somente como um gatilho, um trigger.

Ou seja, a main serve apenas para iniciar o programa, chamando funções.

O comando return - obtendo resultados das funções

Se você tiver notado bem, até o momento, em nosso estudo sobre as funções na linguagem C, as tarefas foram todas realizadas dentro das funções.

E se quiséssemos obter o resultado da soma, que foi calculada internamente da função *soma()*? Ou o resultado do produto? Nem sempre exibir os resultados, como foi feito no exemplo de programa em C passado e quando usamos funções aninhadas, é o que queremos.

Muitas vezes queremos o resultado em si, e é isso que aprenderemos nesse artigo de nossa apostila.

- **O que é e como usar o comando return em C**

Lembra que declaramos as funções com uma palavrinha no começo, *void* ? Pois é, *void* em inglês quer dizer vazio, sem nada. O que isso quer dizer?

Por definição, toda função em C retorna algo, algum valor, variável. Pode retornar um inteiro, um float, um caractere, um vetor, struct ou outro tipo que você criou (veremos tudo isso em nosso curso online de C).

O *void* serviu pra indicar que tais funções não retornavam nada. E é verdade. Elas simplesmente mostravam algo na tela, faziam uns cálculos, mas tudo ficava dentro do escopo das funções. Do lado de fora delas, não tínhamos acesso a esses cálculos.
Era como se as funções fossem caixas-pretas inacessíveis.

Pois bem, poucas coisas em programação são isoladas assim.
Na verdade, os diversos aplicativos e tarefas de uma máquina se comunicam, mandando informação um pro outro. E boa parte dessa comunicação é feita através do retorno das funções.

Se você já estudou outras linguagens, em Java as funções são chamadas de métodos e em Perl, de sub-rotinas. Nomes diferentes, mas ‘função’ parecida.

Se vamos retornar alguma informação de uma função, não vamos mais usar o void.
Vamos usar outros tipos conhecidos.

- **Exemplo de código C**

Crie uma função que recebe dois números como argumentos e retorna a soma deles.

Vamos criar, na *main()*, três variáveis inteiras: duas para pedir ao usuário e uma terceira que irá armazenar o resultado da soma dessas variáveis. Porém, essa soma será feita na função *soma()*.

Declaramos, dentro dessa função, um inteiro ‘c’ que irá armazenar o valor da soma dos parâmetros. Vamos passarmos ‘num1’ e ‘num2’ como argumentos para a função de soma, que sempre armazena esses valores como ‘a’ e ‘b’, pois não importa o nome das variáveis que a função recebe, ela apenas colhe o valor e armazena em ‘a’ e ‘b’.

E a variável que declaramos dentro da função vai receber esse valor inteiro. Como declaramos a função como: *int soma(int a, int b)*, ela tem que retornar um valor inteiro No caso, ‘c’.

Usamos o comando ***return*** para retornar esse valor: ***return c;***
Veja como ficou nosso código C:

```
#include <stdio.h>

int soma(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

int main()
{
    int num1,
        num2,
        result;
```

```
    printf("Numeros a serem somados: ");
    scanf("%d", &num1);
    scanf("%d", &num2);

    result = soma(num1, num2);

    printf("Resultado: %d\n", result);
}
```

- **Exemplo de código**

Escreva uma função que recebe dois floats como argumento, e retorna o produto deles.
Use o mínimo de variáveis possíveis.

No exemplo passado, declaramos a variável ‘c’ e a variável ‘result’ apenas para efeito de organização e didática, pois poderíamos ter feito:
return (a+b);
e
printf(“Resultado: %d\n”, soma(num1,num2));

Veja como fica o código da função que multiplica dois floats:

```
#include <stdio.h>

float produto(float a, float b)
{
    return (a*b);
}

int main()
{
    float num1,
          num2;

    printf("Numeros a serem multiplicados: ");
    scanf("%f", &num1);
    scanf("%f", &num2);

    printf("Resultado: %.2f\n", produto(num1,num2));
}
```

Variáveis Locais - Protótipo de uma função em C

No tutorial passado, de nossa apostila digital C Progressivo, quando estudamos o comando *return* em C, chegamos a declarar uma variável dentro da função.

Outro detalhe que vínhamos fazendo era declarar uma função com parâmetros e invocar tal função com argumentos, mas os nomes das variáveis declaradas nos parâmetros eram sempre diferentes dos nomes das variáveis do argumento.

Vamos entrar em detalhes nesse assunto e estudar a relação entre variáveis declaradas dentro de funções.

Também vamos ensinar uma forma diferente de declarar funções, usando protótipos, que nos ajudam a deixar nosso código mais profissional e organizado.

- **Variáveis Locais em C**

Variáveis locais são variáveis que são declaradas dentro do escopo da função.
Ou seja, são variáveis que não tem nenhuma ligação com o ‘mundo’ fora da função.

Este tipo de variável é chamado de variável local, pois ela só existe dentro da função.
O C cria essa variável durante a execução da função, e após o término desta, ela deixa de existir.
Assim, não é possível acessar ela fora da função.

Vamos usar um exemplo, em que declaramos uma variável chamada ‘teste’ dentro da *main()*, depois passamos essa variável como argumento para uma função que irá modificar seu valor (incrementando), e depois vamos ver o valor dessa variável.

```
include <stdio.h>

void teste(int a)
```

```
{
    a++;
    printf("Estou dentro da funcao! A variavel foi alterada, e aqui dentr vale: a
= %d\n\n",a);
}

int main(void)
{
    int num1 = 1;

    printf("Valor inicial de 'num1': %d\n\n", num1);
    teste(num1);
    printf("Valor de 'num1' apos ir pra funcao: %d\n", num1);
}
```

Ficou claro que as funções não alteram o valor dos argumentos, nesse caso. Isso porque essa passagem de parâmetros é dita ser: **passagem por valor**.

Esse nome se deve ao fato de passarmos uma cópia do argumento para as funções. Não passamos a variável em sim, mas sim uma cópia dela.

Logo, as funções trabalha com uma cópia, com um valor igual ao da variável passada como argumento, e não com a variável em si.

Note que, até o momento, usamos o nome das variáveis de argumento (num1, num2) diferente das variáveis de parâmetro (a, b).

Vamos chamar tudo de ‘a’ e você verá que, mesmo com o nome da variável sendo o mesmo, a função só trabalhar com uma cópia da variável e acaba não alterando, em nada, essa variável:

```
#include <stdio.h>

void teste(int a)
{
    a++;
    printf("Estou dentro da funcao! A variavel foi alterada, e aqui dentr vale: a
= %d\n\n",a);
}

int main(void)
```

```
{
    int a = 1;

    printf("Valor inicial de 'a': %d\n\n", a);
    teste(a);
    printf("Valor de 'a' apos ir pra funcao: %d\n", a);
}
```

- **Como alterar o valor de uma variável passada como argumento**

Se você quiser passar uma variável como argumento para uma função e quer que ela seja alterada, as duas técnicas mais usadas são: **passar o endereço da variável** e **receber o retorno da função**.

Em breve, quando você aprender ponteiros, irá saber a diferença entre passar o valor de uma variável e passar o endereço dessa variável.

Mas, de antemão, adiantamos que se passarmos o endereço da variável, a função irá alterar o valor da variável.

Para passarmos um endereço de uma variável *x* qualquer, colocamos o símbolo ***&*** antes da variável: ***&x***.

Mas, como dissemos, estudaremos isso em detalhes na unidade sobre ponteiros em C.

A outra maneira de alterarmos o valor de uma variável, é fazer com que ela receba o ***return*** de uma função. Veja:

```
#include <stdio.h>

int teste(int a)
{
    a++;
    printf("Estou dentro da funcao! A variavel foi alterada, e aqui dentr vale: a
= %d\n\n",a);
    return a;
}

int main(void)
{
    int a = 1;
```

```
    printf("Valor inicial de 'a': %d\n\n", a);
    a=teste(a);
    printf("Valor de 'a' apos receber o return da funcao: %d\n", a);
}
```

- **Protótipos de uma função em C: o que é e como declarar**

Sempre declaramos as funções antes da *main()*, em nosso curso.
Mas à medida que seu código for ficando complexo, você terá dezenas, centenas ou milhares de linhas de código de funções declaradas antes de ver a *main()*, o que não é muito interessante, já que precisamos ver a *main()* para saber o que cada código em C faz.

Uma técnica usada por bons programadores em C, é a de declarar antes da *main()* apenas o protótipo de cada função, e a função em si é declarada **após** a *main()*.

Para declarar um protótipo, basta pegarmos o cabeçalho de cada função e adicionar um ponto-e-vírgula ao fim deste. E não colocamos colchetes depois.

As funções completas, como vinham declarando até então, será usada após a função *main()*.

É como se, através do uso dos protótipos, disséssemos para o C: “Hey, essa função existe, ok? Mas eu declarei ela somente ao fim da main(). Mas sabia, de antemão, que ela existe, está declarada e vamos usá-la. Portanto, vá lá onde ela está declarada e compile ela!”

O exemplo passado de código, usando protótipos, é:

```
#include <stdio.h>

int teste(int a);

int main(void)
{
    int a = 1;

    printf("Valor inicial de 'a': %d\n\n", a);
```

```
    a=teste(a);
    printf("Valor de 'a' apos receber o return da funcao: %d\n", a);
}

int teste(int a)
{
    a++;
    printf("Estou dentro da funcao! A variavel foi alterada, e aqui dentro vale: a
= %d\n\n",a);
    return a;
}
```

Gerando números aleatórios em C: rand, srand e seed

Você pode nunca ter ficado atento para isso, mas números aleatórios são vitais em praticamente todos os ramos da computação;

Em jogos que você joga contra os inimigos, como eles sempre aparecem em posições e lugares diferentes? Simples, eles aparecem aleatoriamente.

Quando vai jogar Age of Empires, por que o local onde sua civilização começa é sempre diferente? Simples, o local foi escolhido aleatoriamente.

E por aí vai. Se é importante e é muito usado, é óbvio que o curso C Progressivo vai ensinar, e agora, neste tutorial de C de nossa apostila.

- **Números aleatórios em C**

À rigor, não existem números aleatórios em Computação. Computadores não pensam ou sorteiam. Computadores obedecem a comandos. O que eles fazem é o mesmo que seus programas fazem: entram laços, loopings, testes condicionais etc (obviamente, mais complexamente, mas à rigor, fazem o mesmo).

Então, os computadores também usam algoritmos e códigos para gerarem esses números que devem ser, teoricamente, aleatórios. Porém, nunca serão totalmente aleatórios, pois são gerados por código, por funções e tarefas específicas

O que vamos falar, na verdade, é sobre números pseudo-aleatórios. São funções e algoritmos matemáticos complexos, e ainda é um tema bastante utilizado ainda hoje.
Porém, isso foge do objetivo de nossa apostila online de C.

Uma boa tentativa de deixarem os números mais aleatórios possíveis, ‘alimentar’ seu algoritmo com números diferentes. Obviamente, o código de uma função que gera números aleatórios é sempre o mesmo, mas se você fornecer números diferentes, ela vai gerar seqüencias diferentes de números aleatórios.

Esses números que fornecemos são chamados de semente, ou *seed*.
E vamos usar o tempo (sim, o tempo cronológico), como semente.

- **A função *rand()* - Gerando números aleatórios**

Pois bem, agora que você já sabe a importância dos números aleatórios no mundo da programação, e no da computação em geral, vamos iniciar nossos estudos

Vamos iniciar mostrando uma função que é comumente usada.
Porém, não vamos usá-la na maneira padrão, pois como vamos mostrar logo mais, há alguns problemas e ajustes necessários a se fazer.

A função que gera números aleatórios em C é a *rand()*.

Ela gera números entre 0 e RAND_MAX, onde esse RAND_MAX é um valor que pode variar de máquina pra máquina.

Pra usar a função *rand()*, temos que adicionar a biblioteca *time.h* e para saber o valor de RAND_MAX, temos que usar a função *stdlib.h*.

Veja um programa que gera 10 números aleatórios em C:

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

int main(void)
{
   int i;
   printf("intervalo da rand: [0,%d]\n", RAND_MAX);

   for(i=1 ; i <= 10 ; i++)
      printf("Numero %d: %d\n",i, rand());

}
```

- **Alimentando a rand() com a srand() : seed**

Parece tudo ok, não? Veja os números gerados no exemplo passado.

Agora feche e rode o programa novamente. E de novo, de novo...notou? (não vale enrolar! teste mesmo!)

A seqüência gerada é sempre a mesma. Para mudar isso, vamos alimentar a função *rand()* com uma semente, com um número, que é o tempo atual.

Assim, toda vez que rodarmos o programa, a *rand()* pega um número de tempo diferente e gera uma seqüencia diferente.
Para fazer isso, basta usar a função *srand()*, que será responsável por alimentar a *rand().* Fazemos isso adicionando o comando:
srand((unsigned)time(NULL)); antes da *rand()*.

E agora nosso aplicativo em C gera uma seqüencia diferente toda vez que rodamos:

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

int main()
{
    int i;
    printf("intervalo da rand: [0,%d]\n", RAND_MAX);
    srand( (unsigned)time(NULL) );

    for(i=1 ; i <= 10 ; i++)
        printf("Numero %d: %d\n",i, rand());
}
```

- **Escolhendo uma faixa de números aleatórios em C**

Ok, agora você é mestre nas artes de geração aleatória de números em linguagem C.
Porém, seu código gera números entre 0 e RAND_MAX.

E se você quiser gerar entre 0 e 10? Ou entre 1 e mil?
Ou entre 0.00 e 1.00?

Para escolher a faixa de valores vamos usar operações matemáticas, principalmente o operador de módulo, também conhecido como resto da divisão: %

Para fazer com que um número '*x*' receba um valor entre 0 e 9, fazemos:
x = rand() % 10

Para fazer com que um número '*x*' receba um valor entre 1 e 10, fazemos:
x = 1 + (rand() % 10)

Para fazer com que um número '*x*' receba um valor entre 0.00 e 1.00, primeiro geramos números inteiros, entre 0 e 100:
x = rand() % 101

Para ter os valores decimais, dividimos por 100:
x = x/100;

Fácil, não?
Pois bem, com os conhecimentos adquiridos até o momento, já podemos criar um jogo em que nós mesmos poderemos jogar e mandar para os amigos.

Como criar um Jogo em linguagem C

Crie um jogo, em C, que sorteia um número entre 1 e mil.
O jogador deve tentar acertar o número sorteado.

Se ele errar, o programa deve dizer se o número sorteado é maior ou menor que o número que o jogador tentou.

Ao acertar o número sorteado, o programa deverá dizer em quantas tentativas o usuário acertou.

Você consegue pensar na melhor estratégia para esse jogo?

Jogo em C: Adivinhe o número que o computador sorteou

Ok, se chegou até aqui, parabéns novamente, pois estudou bastante coisa!

Que tal agora fazer um jogo em C usando todos os conhecimentos que adquiriu até agora, em nossa apostila online de C?

Embora seja um game simples, você ver que a sensação de criar o próprio game com suas mãos, linha-a-linha irá lhe propiciar uma sensação única.

O curso online e gratuito de C, C Progressivo garante que este será o melhor game que você já jogou!

Como criar um jogo simples em C

Crie um jogo, em C, que sorteia um número entre 1 e mil.
O jogador deve tentar acertar o número sorteado.

Se ele errar, o programa deve dizer se o número sorteado é maior ou menor que o número que o jogador tentou.
Ao acertar o número sorteado, o programa deverá dizer em quantas tentativas o usuário acertou.

Você consegue pensar na melhor estratégia para ganhar esse jogo?

Para criar esse jogo, precisaremos dos conhecimentos de:
Teste condicional IF ELSE
Laço DO WHILE
Funções com uso de parâmetros e argumentos
Protótipo de funções
Funções com return
Números aleatórios em C

Vamos usar um laço DO WHILE para controlar se o jogo deve começar novamente, ou não.

A condição será a variável 'continuar'. Se o jogador não quiser mais jogar, ele digita 0 ou digita qualquer outro número para continuar a jogar (0 é falso e qualquer outro número é true).

Pois bem, decidido que ele vai jogar, precisamos sortear um número, e isso é feito dentro da função 'random()', que não recebe nenhum argumento, apenas retorna um inteiro entre 1 e mil.
Vamos armazenar o número sorteado na variável 'password', será a nossa senha.

Fazemos com que a variável 'attempt' receba valor 0, pois é ela que vai contar quantas tentativas o jogador fez até acertar a senha.

Ele tentará adivinhar o número, e essa entrada que ele der será armazenada na variável 'number'.
Após tentar acertar, vamos checar as dicas.
Mandamos as variáveis 'number', 'password' e 'attemp' para a função 'dicas()'.

Se o número 'number' que fornecemos for maior que a senha, a dica diz que nosso número está maior.
Se o 'number' for menor que a senha, a dica nos mostra que nossa tentativa foi abaixo do número correto.

Ora, se não for maior nem menor, é porque acertamos a senha, e a dica nos parabenizará e mostrará em quantas tentativas acertamos.

Quando a rodada acaba? Ora, quando o jogador acertar a senha!
Enquanto ele não acertar, ele ficará preso dentro do looping, que no caso é um novo laço DO WHILE, onde a condição para sair é acertar a senha.

Enquanto não acertar, a condição 'number != password' será verdadeira.

Assim, o código completo de nosso primeiro jogo em C fica:

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

int random();
void dicas(int number, int password, int attempt);
```

```c
int main()
{
    int continuar=1,
        password,
        attempt,
        number;

    do
    {
        system("cls || clear");
        printf("Sorteando numero entre 1 e 1000...\n");
        password = random();

        printf("Comecou! Tente adivinhar o numero!\n\n");
        attempt = 0;

        do
        {
            attempt++;
            printf("Tentativa %d: ", attempt);
            scanf("%d", &number);
            dicas(number,password,attempt);
        }
        while( number != password);

        printf("Digite 0 para sair, ou qualquer outro numero para continuar: ");
        scanf("%d", &continuar);
    }
    while(continuar);

}

int random()
{
    srand( (unsigned)time(NULL) );

    return (1 + rand()%1000);
}

void dicas(int number, int password, int attempt)
{
    if(number > password)
            printf("O numero sorteado e menor que %d\n\n", number);
    else
```

```
        if(number < password)
            printf("O numero sorteado e maior que %d\n\n", number);
        else
            printf("Parabens! Voce acertou o numero em %d tentativas!\n\n",
attempt);
}
```

A estratégia para acertar é sempre ir na metade. Primeiro tente o número 500.
Assim, você eliminará metade das possibilidades, pois já sabe que vai estar entre 1 e 499, ou entre 501 e 1000.

Se for maior que 500, tente 750. Se for menor, tente 250.
A explicação disso é eliminar metade (250), dos 500 possíveis números...depois 125, depois 62, depois 31, depois 15, depois 8, depois 4, depois 2 e por fim, 1. Ou seja, sempre se acerta, em no máximo 9 tentativas.

Gostou? Coloque seu nome

"Desenvolvido por Joãozinho - Curso C Progressivo: www.cprogressivo.net" e envie para seus amigos.

Recursão

O título desse artigo em C pode parecer estranho, à primeira vista, mas ao fim do tutorial você entenderá essa ‘piada interna’ entre os programadores.

Neste tutorial de nossa apostila de C, vamos ensinar uma importante técnica: a recursão.

- ## Recursividade em C

Uma função que é dita recursiva é aquela que invoca ela mesma.
Já explicamos, e demos exemplos, que é possível, recomendável e normal que uma função invoque outra.

Porém também é possível que uma função chame ela mesma, mas é preciso um cuidado especial para não cairmos em um looping infinito.

Geralmente, para que uma função não fique invocando ela mesma indefinidamente, devemos fazer umas alterações no argumento, ao invocar novamente a função ao passo que devemos definir, na função, testes condicionais sobre o parâmetro para saber onde devemos parar de invocar a função.

Ok, a equipe do curso C Progressivo assume: essa explicação é realmente complicada e todos sentiram dificuldade a primeira vez que a leram.

Mas, como de costume, vamos apresentar diversos exemplos de códigos, bem comentados, para que você possa entender melhor isso na prática.

- ## Exemplo de código usando Recursão

Crie um programa em C que peça um número inteiro ao usuário e retorne a soma de todos os números de 1 até o número que o usuário introduziu ou seja: 1 + 2 + 3 + ... + n
Utilize recursividade.

Vamos criar uma função *soma(int n)*.
Se n=5, essa função deve retornar: soma(5) = 5 + 4 + 3 + 2 + 1
Se n=4, essa função deve retornar: soma(4) = 4 + 3 + 2 + 1
Se n=3, essa função deve retornar: soma(3) = 3 + 2 + 1

E assim por diante. Porém, para fazermos uso da brilhante idéia matemática da recursividade, temos que notar padrões.
Veja que:
soma(5) = 5 + 4 + 3 + 2 +1 = 5 + soma(4)

O mesmo para:
soma(4) = 4 + 3 + 2 + 1 = 4 + soma(3)

Ou seja, temos a fórmula geral:
soma(n) = n + soma(n-1)
Concorda?

Ou seja:
soma(n) = n + soma(n-1) = n + (n-1) + soma(n-2) = n + (n-1) + (n-2) + soma(n-3)...

E onde essa soma para? Para quando o último elemento dessa soma for 1.
Então:
soma(n) = n +(n-1) + (n-2) + (n-3) + ... + 1

Agora vamos traduzir essa fórmula em termos de programação.
A função recebe um número, e a primeira coisa que ela deve fazer é ver se esse valor é 1.
Se for, deve retornar 1, afinal:
soma(1) = 1

E se não for 1, deve retornar:
n + soma(n-1)

Isso é feito da uma maneira muito simples, através de um simples teste condicional do tipo IF ELSE.
Veja como ficou nosso código em C:

```
#include <stdio.h>

int soma(int n)
{
    if(n == 1)
        return 1;
```

```
    else
        return ( n + soma(n-1) );
}

int main()
{
    int n;
    printf("Digite um inteiro positivo: ");
    scanf("%d", &n);

    printf("Soma: %d\n", soma(n));
}
```

- **Exemplo de código usando recursão em C**

Crie um programa que calcule o fatorial de um número fornecido pelo usuário através da recursividade.
O fatorial de 'n' é representado por n!, onde:
n! = n * (n-1) * (n-2)*..1

O raciocínio desse exemplo é análogo ao do exemplo anterior, porém, vamos usar a multiplicação ao invés da soma.
Antes de resolvermos, vamos ver a idéia matemática por trás do fatorial.

Como dissemos na questão, para n=5:
5! = 5 * 4 * 3 * 2 * 1

Para n=4:
4! = 4 * 3 * 2 * 1

Para n=3:
3! = 3 * 2 * 1
E assim sucessivamente.
Porém, note que:
5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!

E também:
4! = 4 * 3 * 2 * 1 = 4 * 3!

Podemos formular uma fórmula geral:

n! = n * (n-1)!

Abrindo esse produto, devemos parar somente quando o último elemento do produto for 1:
n! = n * (n-1)! = n * (n-1) * (n-2)! = n * (n-1) * (n-2) * ... * 1

Para programar isso, criamos uma função *fatorial(int n)* que retorna 1 se for passado 1 como argumento (pois *fatorial(1) = 1*) e caso o argumento seja maior que 1:
fatorial(n) = n * fatorial(n-1)

Isso, assim como no exemplo passado, e na maioria das funções usando a idéia de recursividade, é resolvido com um simples teste condicional IF ELSE.
Veja como ficou nosso código em C que calcula o fatorial:

```
#include <stdio.h>

int fatorial(int n)
{
    if(n == 1)
        return 1;
    else
        return ( n * fatorial(n-1) );
}

int main()
{
    int n;
    printf("Digite um inteiro positivo: ");
    scanf("%d", &n);

    printf("%d! = %d\n", n, fatorial(n));
}
```

Ou seja, pra calcular a função *soma()* é preciso usar a função *soma()*.
Pra calcular o fatorial com a função *fatorial()* é preciso usar a função *fatorial()*.

Exercícios sobre funções em C

Resolva as seguintes questões usando conceitos funções, de return, de números aleatórios e paramêtros/argumentos:

0. Crie uma função que receba um valor e informe se ele é positivo ou não.

1. Crie uma função que receba um valor e diga se é nulo ou não.

2. Crie uma função que receba três valores, 'a', 'b' e 'c', que são os coeficientes de uma equação do segundo grau e retorne o valor do delta, que é dado por 'b^2 - 4ac'

3. Usando as 3 funções acima, crie um aplicativo que calcula as raízes de uma equação do 2o grau:
$ax^2 + bx + c=0$
Para ela existir, o coeficiente 'a' deve ser diferente de zero.
Caso o delta seja maior ou igual a zero, as raízes serão reais. Caso o delta seja negativo, as reais serão complexas e da forma: x + iy

4. Crie uma função em linguagem C que receba 2 números e retorne o maior valor.

5. Crie uma função em linguagem C que receba 2 números e retorne o menor valor.

6. Crie uma função em linguagem C que receba 3 números e retorne o maior valor, use a função da questão 4.

7. Crie uma função em linguagem C que receba 3 números e retorne o menor valor, use a função da questão 5.

8. Crie uma função em linguagem C chamado Dado() que retorna, através de sorteio, um número de 1 até 6.

9. Use a função da questão passado e lance o dado 1 milhão de vezes. Conte quantas vezes cada número saiu.
A probabilidade deu certo? Ou seja, a porcentagem dos números foi parecida?

10. Crie um aplicativo de conversão entre as temperaturas Celsius e Farenheit. Primeiro o usuário deve escolher se vai entrar com a temperatura em Célsius ou Farenheit, depois a conversão escolhida é realizada através de um comando SWITCH.
Se C é a temperatura em Célsius e F em farenheit, as fórmulas de conversão são:

C= 5.(F-32)/9
F= (9.C/5) + 32

11. Um professor, muito legal, fez 3 provas durante um semestre mas só vai levar em conta as duas notas mais altas para calcular a média.
Faça uma aplicação em C que peça o valor das 3 notas, mostre como seria a média com essas 3 provas, a média com as 2 notas mais altas, bem como sua nota mais alta e sua nota mais baixa.

Desafio 1: Programe um aplicativo em C que acha todos os números primos até 1000
Número primo é aquele que é divisível somente por 1 e por ele mesmo.

Desafio 2: Programe um aplicativo em C que recebe dois inteiros e retorna o MDC, máximo divisor comum.

Desafio 3: Programe um aplicativo em C que ache todos os números perfeitos até 1000.
Número perfeito é aquele que é a soma de seus fatores. Por exemplo, 6 é divisível por 1, 2 e 3 ao passo que 6 = 1 + 2 + 3.

Desafio 4: Crie um programa em C que receba um número e imprima ele na ordem inversa.
Ou seja, se recebeu o inteiro 123, deve imprimir o inteiro 321.

Soluções

Questão 0, 1, 2 e 3:
Programa que calculas raízes de equação do segundo grau, até as complexas, em C
As questões 0, 1, 2 e 3 vão ensinar você a construir, passo a passo, um programa que recebe os coeficientes de uma equação do 2º grau, e devolver as raízes dele.

As funções que checam se um número é positivo, se é nulo e que retornam o valor delta são óbvias e você deve entender facilmente como fazer.

Vamos declarar os coeficientes como float, bem como as raízes, para que nossa aplicação seja mais flexível e aceite números decimais.

Como de praxe, vamos criar um laço DO WHILE para saber se o usuário desejar rodar o programa novamente, para calcular as raízes de uma nova equação. Como de costume, também usaremos uma variável inteira '*continuar*' para o usuário escolher se vai querer continuar no programa ou não.

A primeira coisa que fazemos dentro do DO WHILE é pedir ao usuário os valores dos coeficientes.

Agora vamos a parte matemática da questão.
Para ser uma equação do segundo grau é necessário que o coeficiente de x^2 seja diferente de 0, no caso, é o nosso '*a*'.
Vamos criar um teste condicional do tipo IF ELSE e a função *nulo()* que criamos, para saber se esse coeficiente é 0 ou não.
Se for, a função *nulo()* retorna 1 e nossa aplicação é encerrada, pois não podemos calcular as raízes nesse caso.

Caso não seja nulo, a função *nulo()* retorna 0 e nosso teste cai no ELSE.
Ok, dentro do ELSE vamos checar se o delta é maior/igual a 0, pois se for, as raízes da equação serão reais.
Nesse caso, simplesmente aplicamos a fórmula de Bháskara.

Porém, vamos necessitar calcular uma raiz quadrada. Isso pode ser feito pela função *sqrt()* (de *square root)*, que está na biblioteca *math.h*, por isso adicionamos ela no início.

Se o delta for menor que 0, as raízes serão imaginárias e serão da forma: *x* + *i.y*, onde *i* é conhecido como número imaginário, *x* é a parte real e *y* é a parte imaginária.
Lembre-se que você não pode calcular a raiz quadrada de um número negativo, por isso vamos multiplicar o delta por -1 para então tirar a raiz quadrada do delta.
Assim, imprimimos os números complexos dividindo a parte real da imaginária.

Esse programinha em C é bem útil, e com certeza você tem um irmão mais novo ou vizinho que irá adorar ter esse programa.

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

int positivo(float n);
int nulo(float n);
float delta(float a, float b, float c);

int main()
{
    float a, b, c,
    raiz1, raiz2;
    int continuar;

    do
    {
        system("cls || clear");
        printf("Programa que acha as raizes de uma equacao do 2o grau: ax^2
+ bx + c = 0\n\n");
        printf("Entre com os coeficientes a, b, e c: ");

        scanf("%f", &a);
        scanf("%f", &b);
        scanf("%f", &c);

        if( nulo(a))
            printf("'a' deve ser diferente de zero.");
        else
        {
            //Se delta for maior ou igual a zero, as raizes sao reais
            if( positivo( delta(a,b,c) ) || nulo( delta(a,b,c) ) )
            {
```

```
            raiz1 = ( (-1)*b + sqrt(delta(a,b,c)) )/(2*a);
            raiz2 = ( (-1)*b - sqrt(delta(a,b,c)) )/(2*a);
            printf("Raiz 1: %.2f\nRaiz 2: %.2f", raiz1, raiz2);
        }
        else
        {
            //Precisamos exibir a parte real e a parte imaginária
separadamente
            //raiz 1
            printf("%.2f + i.%.2f\n", (-1)*b/(2*a),
                sqrt((-1)*delta(a,b,c))/(2*a) );

            //raiz 2
            printf("%.2f - i.%.2f\n", (-1)*b/(2*a),
                sqrt((-1)*delta(a,b,c))/(2*a) );
        }
    }

    printf("\n\nCalcular mais raizes?\n1. Continuar\n0. Sair\n");
    scanf("%d", &continuar);
  }
  while(continuar);
}


int positivo(float n)
{
  if(n > 0)
    return 1;
  else
    return 0;
}

int nulo(float n)
{
  if(n == 0)
    return 1;
  else
    return 0;
}

float delta(float a, float b, float c)
{
  return ((b*b) - (4*a*c));
}
```

4. Crie uma função em linguagem C que receba 2 números e retorne o maior valor.

```
float maior2(float num1, float num2)
{
   if(num1 >= num2)
      return num1;
   else
      return num2;
}
```

5. Crie uma função em linguagem C que receba 2 números e retorne o menor valor.

```
float menor2()2(float num1, float num2)
{
   if(num1 <= num2)
      return num1;
   else
      return num2;
}
```

6. Crie uma função em linguagem C que receba 3 números e retorne o maior valor, use a função da questão 4.

```
float maior3(float num1, float num2, float num3)
{
   if( maior2(num1, num2) >= num3)
      return (maior2(num1, num2));
   else
      return num3;
}
```

7. Crie uma função em linguagem C que receba 3 números e retorne o menor valor, use a função da questão 5.

```
float menor3(float num1, float num2, float num3)
{
   if( menor2(num1, num2) <= num3)
      return (menor2(num1, num2));
   else
      return num3;
```

```
}
```

8. Crie uma função em linguagem C chamado Dado() que retorna, através de sorteio, um número de 1 até 6.

```
int dado()
{
   return (1 + rand()%6);
}
```

9. Use a função da questão passado e lance o dado 1 milhão de vezes. Conte quantas vezes cada número saiu.
A probabilidade deu certo? Ou seja, a porcentagem dos números foi parecida?

Vamos criar 6 variáveis do tipo inteiro para armazenar quantos vezes cada um dos números do dado foi sorteado (não esqueça de inicializar elas com 0).

A seguir, faça um looping de 1 milhão de iterações.
A cada iteração, use um teste condicional para saber que número foi gerado pela função*dado()*, e incremente a variável correspondente.

Ao final do laço você saberá quantas vezes cada número foi sorteado. Divida por 1 milhão para saber a parcela que cada número apareceu, ou multiplique por 100 depois para obter a porcentagem.

O resultado deve ser algo bem 'justo', com cerca de 16,7% de aparição para cada face do dado.

```
#include <stdio.h>

#include <time.h>

int dado();

int main()
{
   int num1=0, num2=0, num3=0, num4=0, num5=0, num6=0,
      count;

   for(count=0 ; count < 1000000 ; count++)
      switch( dado() )
      {
         case 1:
```

```
            num1++;
            break;
        case 2:
            num2++;
            break;
        case 3:
            num3++;
            break;
        case 4:
            num4++;
            break;
        case 5:
            num5++;
            break;
        case 6:
            num6++;
    }

    printf("Numero 1: %d -> %.2f\%%\n", num1, (num1/1000000.0)*100);
    printf("Numero 2: %d -> %.2f\%%\n", num2, (num2/1000000.0)*100);
    printf("Numero 3: %d -> %.2f\%%\n", num3, (num3/1000000.0)*100);
    printf("Numero 4: %d -> %.2f\%%\n", num4, (num4/1000000.0)*100);
    printf("Numero 5: %d -> %.2f\%%\n", num5, (num5/1000000.0)*100);
    printf("Numero 6: %d -> %.2f\%%\n", num6, (num6/1000000.0)*100);

    printf("\nTotal: %d -> %.2f\%%\n",
num1+num2+num3+num4+num5+num6,
                        (num1/1000000.0)*100+(num2/1000000.0)*100+
                        (num3/1000000.0)*100+(num4/1000000.0)*100+
                        (num5/1000000.0)*100+(num6/1000000.0)*100);

}

int dado()
{
    return (1 + rand()%6);
}
```

Vetores

Nos artigos a seguir, estudaremos os importantes conceitos de Vetores, que são uma introdução ao estudo das estrutura de dados.

A partir de agora, vamos poder criar, analisar, alterar, salvar e extrair informações de dezenas, centenas ou milhares de dados (variáveis) de uma maneira mais automatizada e programada.

O estudo dos vetores constitui a base para a próxima seção do curso online e gratuito C Progressivo, sobre o estudo das strings, e serão essenciais para o entendimento dos ponteiros em C.

Não tenha pressa. Estude, leia, releia, tire suas dúvidas, mas estude com calma os vetores.

O que são vetores, como declarar e quando usar

Dando início a mais uma importante unidade em nosso curso online e gratuito de C, vamos iniciar nossos estudos sobre as estruturas de dados.

Neste artigo inicial da seção de vetores de nossa apostila, vamos estudar o tipo de estrutura de dado mais simples: **os vetores**, também conhecidos por **arrays**.

- ## O que é um vetor em C... E para quê serve?

Imagine que você foi contratado para criar um programa em C para uma escola.
Nesse programa você tem que armazenar as notas dos alunos, nomes, médias, nome dos pais, faltas e tudo mais.

E aí? Vai declarar quantos inteiros pra armazenar as notas? Centenas? Milhares?
E quantos caracteres para armazenar esses nomes?
E quantos floats para armazenar as notas e médias, de cada matéria, para cada aluno?

É quase que humanamente impossível fazer isso. Mas não se preocupe, pois você programa em C e tem total domínio de sua máquina, você não perderá tempo declarando inúmeras variáveis, fará com que o computador faça isso pra você.

E é disso que se trata estrutura de dados: estudar, manipular, organizar, salvar e extrair informações de uma grande quantidade de dados. Automatizar esse processo para que não precisemos declarar, inicializar e fazer outras operações em nossas variáveis de um modo manual.

- ## Como declarar um vetor em C

Agora que já sabe para que serve e como são importantes, vamos começar a usar os vetores/arrays em linguagem C.

A sintaxe é a seguinte:
tipo nome[numero_de_elementos];

Ou seja, a sintaxe é a mesma de declarar uma variável normal, mas não vamos declarar somente uma, vamos declarar várias. E o par de colchetes ao lado do nome da variável serve para isso: especificar quantas daquelas variáveis estamos declarando.

Por exemplo, vamos declarar 10 inteiros que vão representar a idade de 10 pessoas:
int idade[10];

Agora 50 floats que vão representar a nota de 50 alunos:
float notas[50];

Bem simples.

A contagem dos índices começa sempre do 0

Embora tenhamos declarado as variáveis com um nome, elas não podem ter um mesmo nome. Por isso, um número é associado ao seu nome.

No caso da *idade[10]*, as variáveis inteira são:
idade[0], idade[1], idade[2], ..., idade[9]
Isso mesmo, o primeiro elemento **é sempre o zero.**
Por isso as lições do curso C Progressivo começam do número 0.
Por isso as questões começam da questão de número 0.

Mais uma vez, isso é muito básico e importante: a contagem, em linguagem C, começa do 0.

No caso das *notas[50]*, as variáveis do tipo float são:
notas[0], notas[1], notas[2], ... , notas[48] e notas[49]

Então, se uma variável tem ‘n’ elementos, seus índices variam, **sempre**, de 0 até n-1, totalizando ‘n’ elementos.

Como usar acessar os elementos de um vetor em C

Declaramos várias variáveis com o mesmo nome, mas como se referir, individualmente, a cada uma delas?
A resposta é simples: usando números, ou índices.

'notas' é um vetor de floats.
Se quiser usar um tipo float, use a seguinte sintaxe: nome[indice]

Então, suas variáveis, de forma independente, são chamadas de: notas[0]. notas[1], notas[10] etc.
Esses serão seus nomes. Você pode usar como usaria as variáveis (na verdade elas são variáveis, como se tivessem sido declaradas manualmente), por exemplo:

Armazenar a nota de um aluno que tirou 10
nota[10]= 10.0 //esse programa em C

Somar a nota de dois alunos:
float soma = nota[3] + nota[4];

Incrementar:
nota[5]++;

Enfim, pode fazer tudo. São variáveis do tipo float normais.
A diferença é que os nomes das variáveis têm números, que são chamados, em programação, de índice, que são criados automaticamente quando você declara um bloco de vários elementos (vulgo vetores, ou arrays).

Como de costume, para fixar melhor, vamos aos exemplos de código!

Exemplo 1: Faça um programa que peça 3 números inteiros ao usuário, armazene em um vetor, depois mostre o valor de cada elemento do vetor, assim como seu índice.

Primeiro declaramos um vetor de inteiros, contendo 3 elementos:
int numbers[3];
Agora vamos pedir pro usuário preencher esses três números.

Lembre-se que você é programador e sabe que os índices vão de 0 até 2. Mas o usuário não. Pro leigo, é número 1, número 2 e número 3, não inicia no 0.

No laço for, o nosso 'indice' vai de 0 até 2.
Porém, ao recebermos o valor de índice 'indice', estamos pedindo ao usuário

o valor do número 'indice+1'.

Por exemplo, para armazenar um valor no 'number[0]', vamos pedir o número '0+1' ao usuário.
Para armazenar um valor no 'number[1]', vamos pedir o número '1+1' ao usuário.
Para armazenar um valor no 'number[2]', vamos pedir o número '2+1' ao usuário.

Usaremos outro laço for para exibir o valor dos números, através dos índices, que variam de 0 até 2.
Porém, novamente, temos que mostrar 1 ao 3 pro cliente, pois pra ele não faz sentido 'número 0 -> valor 10 ' e sim 'número 1 -> valor 10'.

Então, nosso código em C fica:

```
#include <stdio.h>

int main()
{
    int number[3],
        indice;

        for(indice=0 ; indice <= 2 ; indice++)
        {
            printf("Entre com o numero %d: ", indice+1);
            scanf("%d", &number[indice]);
        }

        for(indice=0 ; indice <= 2 ; indice++)
            printf("Numero %d = %d\n", indice+1, number[indice]);

}
```

Exemplo: Faça um programa em C que peça ao usuário duas notas que ele tirou e mostre a média.
Use vetores! Aliás, use somente um vetor para essas três variáveis.

Se não fosse pelo uso do vetor, você faria esse exemplo com o pé nas costas e dançando salsa.

Mas tem que usar um vetor.

Vamos declarar um vetor de float de três elementos.

Nas duas primeiras posições armazenamos as notas do usuário (nota[0] e nota[1]), e na terceira posição (nota[2]) armazenaremos a média (nota[0] + nota[1])/2.

Veja como ficou nosso código C:

```
#include <stdio.h>

int main()
{
    float notas[3];

    printf("Insira sua primeira nota: ");
    scanf("%f", &notas[0]);

    printf("Insira sua segunda nota: ");
    scanf("%f", &notas[1]);

    notas[2] = (notas[0] + notas[1])/2;

    printf("Sua media e: %.2f\n", notas[2]);
}
```

Sim, muito simples. São simplesmente variáveis normais, a diferença é que são várias delas.

É até mais fácil trabalhar com vetores do que com variáveis declaradas manualmente, por conta dos índices.

Afinal, números tem uma ordem lógica, já os nomes que damos as variáveis não.

Até fazemos uma ressalva, evite usar variáveis como: a, b, c, x, y, i, j... ou a primeira letra que você encontrou.

Use 'linha', 'coluna', 'indice', 'contador', 'count', 'media', 'nota', 'continuar' etc, como fizemos aqui.

Pode parecer perda de tempo usar uma variável com nome tão grande. Mas é porque ainda estamos estudando.

Quando você for profissional e já tiver feito programa com milhares de linhas de código, ficará impossível entender o que é sopa de letras que você usou como nome de variáveis.

Mas se tiver usado nomes que façam sentido, você e outros programadores também entenderão de maneira mais simples.

Não é um conselho, é um diferencial.

Inicializando vetores – Vetor de caracteres e Lixo

Agora que você aprendeu o que são vetores, como declarare viu alguns exemplos de seu uso, vamos explicar um pouco mais sobre a inicialização de vetores e os cuidados que devemos ter ao manipular vetores, ou arrays, em C.

Aproveitando o assunto sobre cuidados com vetores, vamos mostrar neste tutorial de nossa apostila de C o que são os 'lixos' que ficam nos vetores (e variáveis), como nos vetores de caracteres.

- **Inicializando vetores**

Assim como nas variáveis, podemos inicializar os vetores assim que declaramos.
Como são vários valores, temos que colocar todos esses valores entre chaves {}.

Veja, pra inicializar um vetor de 3 inteiros:

```
int numeros[3] = { 1, 2, 3};
```

Ou seja:

```
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
```

E para caracteres? Ora, vale o mesmo.
Vamos armazenar a frase: C Progressivo
Com o caractere de new line \n, temos que declarar 14 caracteres (incluindo o espaço em branco, que também é um char em C).

Para exibir, nossa programa ficaria assim:

```
#include <stdio.h>

int main()
{
   char curso[14] = {'C', ' ', 'P', 'r', 'o', 'g', 'r'
                   , 'e', 's', 's', 'i', 'v', 'o','\n'};
   int indice;
```

```
    for(indice=0 ; curso[indice] != '\n' ; indice++)
        printf("%c", curso[indice]);
}
```

Consegue, só olhando o código, saber o que vai acontecer?

Mas um vetor de caracteres é uma coisa especial, em linguagem C, chamada *string*, e teremos uma seção em nosso curso especialmente dedicada ao estudo das *strings*, de tão importantes que elas são.

- **O que é o Lixo, em linguagem C**

Já se perguntou por que declaramos variáveis? Aliás, o que é declarar?
O ato de declarar é sinônimo de alocar. Quando declaramos uma variável, estamos alocando (separando) um espaço na memória do computador para ela.

Lembra que dissemos que é uma linguagem de baixo nível, que trabalha no ‘talo’ da sua máquina?
Pois é, declarar é como dizer: “Ei C, separa um pedaço da memória pra essa variável, ok?”

Mas nos espaços da memória já existem ‘coisas’, números, bits, dados...enfim, todo bloquinho da memória do seu computador já foi usando antes, por outros programas que deixaram ‘vestígios’.

Diferente de outras linguagens, como Java que cada variável inicie com valor zero, ‘’ ou NULL
, o C não limpa esses espaços. É um erro muito comum dos iniciantes não inicializar suas variáveis, então quando vão rodar seus programas percebem uns números gigantes e/ou negativos, totalmente nada a ver.

Vamos captar alguns lixos!
Declare um vetor, de quantas posições quiser de 10, por exemplo.
Não inicie nada, vamos ver que valores estavam armazenados antes do C alocar esses espaços para nosso aplicativo.

Para isso, basta rodar um laço que percorra todos as casas e verá o tanto de coisas bizarras que aparecem:

```c
#include <stdio.h>

int main()
{
    int lixo[10];
    int indice;

    for(indice=0 ; indice < 11 ; indice++)
        printf("Lixo na posicao %d: %d\n", indice, lixo[indice]);
}
```

No caso, até extrapolamos os valores do índice do vetor, e fomos até a posição 10.
Podemos fazer isso pois quando o C seleciona um espaço de memória pra vetores, ele tenta sempre alocar espaços contíguos (vizinhos).

Teste com caracteres também:

```c
#include <stdio.h>

int main()
{
    char lixo[10];
    int indice;

    for(indice=0 ; indice < 11 ; indice++)
        printf("Lixo na posicao %d: %c\n", indice, lixo[indice]);
}
```

Interessante, não?
Como já dizia o filósofo programador C: há mais coisas entre o programa e o hardware que nossa vã programação imagina.

Não use números, use constantes: const e #define

Como saber se uma pessoa já pode ser presa? Fácil, pela idade:

if(idade>=18) ou if(idade>17)

Como saber se uma pessoa já pode votar? Fácil, pela idade:
if(idade>=16) ou if(idade>15)

Parece certo, não é? A rigor, está sim.
Está correto, mas absolutamente não recomendável. Vamos explicar isso neste tutorial de C de nossa apostila.

- **Por que se deve evitar usar números em programação C**

Imagine que você foi contratado para fazer um programa para o governo federal.

Dentre outras coisas, seu aplicativo C deve dizer ao cidadão se ele está apto a dirigir, votar, se alistar nas Forças Armadas, doar sangue, viajar sozinho, idade pro homem se aposentar, pra mulher, tempo de serviço e milhares de outras coisas.

Facilmente, em seu código, você iria ultrapassar os milhares de testes condicionais, iria usar várias idades: 14, 16, 18, 21, 60, 65 e outras.
Isso pra você não é problema e faz todo sentido. E em um projeto profissional, é comum um programa ter dezenas de milhares de linhas de código.

Agora vamos mostrar os riscos do uso desses números.
Imagine que o governo mudou a maioridade penal: agora jovens de 16 anos podem ser presos.

Devido aos acidentes, somente pessoas com 21 anos, ou mais, podem comprar bebida alcoólica.

E como o país está crescendo e as pessoas melhoraram seu nível profissional, podem se aposentar com menos tempo de serviço.

E agora, José? Sair vasculhando milhares de testes condicionais, loopings, funções, bibliotecas em milhares de linhas de código, procurando esses números?

“Ah, vai em SEARCH e REPLACE ALL”. Errado, nem tudo deve mudar. Nem todos os números ‘18’ (no caso da maioridade penal) vão mudar pra ‘16’ (pra se alistar, continua 18).

Pode ter certeza, isso iria levar semanas e é quase impossível fazer isso sem erros, sem ter deixado passar algo ou ter mudado algo que não devia. As complicações são muitas, e o ato de usar esses números em uma aplicação grande é irresponsabilidade, embora seja um erro catastrófico e básico, infelizmente muitos programadores que são profissionais ainda cometem esse tipo de barbaridade. Concordo com você, deviam ser queimados em praça pública (brincando).

Mas não se preocupe você estudou pelo curso C Progressivo, e caso tenha lido nossas lições e feitos os exercícios, garantimos sua competência.

- **Definindo e usando constantes em C**

Existem, basicamente, duas maneiras de definir constantes em linguagem C: Através do comando *const* e da diretiva de pré-processamento: #define

Um costume dos programadores C é declarar essas constantes no começo do programa, depois dos #include .

A sintaxe para declarar constantes usando a palavra reservada const é:
const tipo_da_variavel nome_da_variavel = valor_da_variavel;

Ou seja, é como declarar uma variável qualquer, como vínhamos fazendo. A diferença é a palavra *const* no início, e o fato de termos que inicializar a variável.

É como se tivéssemos definido uma variável, e de fato é alocado um espaço em memória para essa constante, que será acessado sempre que usarmos essa variável.

O que é diferente do *#define*, que na verdade não faz parte da linguagem C, é algo que vem antes, é um comando pro pré-processador que ocorre antes da compilação.

A sintaxe para usar o #define é a seguinte:
#defina nome_da_variavel valor_da_variavel

Isso mesmo, sem símbolo de igual, sem o tipo de variável e sem ponto-e-vírgulo.
É como se disséssemos ao C: “Hey C, troque cada aparição da palavra ‘nome_da_variavel’ por ‘valor_da_variavel’, e só depois compile”.

Exemplo de código:
Crie um aplicativo em C que peça ao usuário sua idade, e diga se ele já pode dirigir (se tiver 16 anos ou mais), se alistar (se tiver 18 anos ou mais) e se já pode se aposentar(65 anos ou mais).
Use constantes: *const* e #define.

```
#include <stdio.h>
#define MAIORIDADE 18
const int aposentadoria = 65;
const int motorista = 16;

int main()
{
    int idade;
    printf("Digite sua idade: ");
    scanf("%d", &idade);

    if( idade >= MAIORIDADE )
        printf("Voce ja pode se alistar e dirigir.\n");
    else
        if( idade >= motorista )
            printf("Voce pode dirigir, mas nao pode se alistar.\n");
        else
            printf("Voce nao pode dirigir nem se alistar\n");

    if( idade >= aposentadoria )
        printf("Voce ja pode se aposentar!\n");
}
```

Note que na #define usamos letras maiúsculas, pois é um costume entre programadores C.

Vetores multidimensionais (Matrizes) em C: vetor de vetores

Agora que a apostila C Progressivo mostrou como declarar,inicializar e usar vetores de vários tipos de variáveis, vamos ensinar a usar os vetores para armazenar outros vetores.

Isso mesmo, vetor com vetores dentro. No mundo real, são conhecidos por tabelas.

- **Vetores multidimensionais, ou Matrizes: o que são e para quê servem**

Nos exemplos passados, sobre vetores, o portal C Progressivo vinha usando somente um tipo de vetor: vetores unidimensionais.

Nossos vetores, ou arrays, tinham uma linha e várias colunas.
Seja ‘n’ o número de colunas, nossos vetores eram matrizes 1 x n.
Sim, uma linha de elementos também é uma matriz.

Vamos agora aprender para que servem e como trabalhar com vetores de mais de uma linha.
Para isso, vamos voltar ao exemplo do colégio, onde declaramos um vetor para armazenar as notas de um aluno.

Na escola, esse aluno tem várias matérias: Matemáticas, Física, Química, Biologia etc.
Vamos supor que existam 5 provas ao longo do ano, para cada matéria.
Poderíamos representar as notas de cada aluno da seguinte maneira:

```
float notasMatematica[5];
float notasFisica[5];
float notasQuimica[5];
```

- **Como declarar e trabalhar com Matrizes em C**

Existe, porém, uma maneira bem mais fácil, lógica e organizada de trabalhar com vários vetores. Quando nós falarmos de vetor, lembre de uma linha com vários elementos.

Pois bem, uma maneira melhor de ver esses vetores de notas, seria na forma de uma tabela.
Uma linha tem as notas de Matemática, na outra as notas de Física e assim vai.
Cada coluna representa as provas feitas: prova 1, prova 2, ... etc.

Vamos fazer isso!

Por exemplo, vamos supor que ele tirou as seguintes notas em Matemática (é uma matriz 1x5):

8.0	7.5	8.5	9.0	8.0

Agora vamos representar as notas em Física, abaixo das de Matemática.
Teremos uma matriz 2x5, ou seja, uma matriz de duas linhas e 5 colunas:

8.0	7.5	8.5	9.0	8.0
8.9	9.0	8.6	8.4	8.0

Agora vamos representar as notas de Química, abaixo das notas de Física.
Teremos uma matriz 3x5, ou seja, uma matriz de três linhas e 5 colunas:

8.0	7.5	8.5	9.0	8.0
8.9	9.0	8.6	8.4	8.0
6.8	7.1	7.0	7.6	6.5

Ok, agora vamos partir para a programação e ver como declarar e passar isso pra linguagem C.

Para declarar a matriz 2x5, fazemos:
float notas[2][5];

Note que temos duas linhas: notas[0][] e notas[1][], e em cada linha dessa temos 5 elementos.

Ou seja, é uma matriz de duas linhas e cinco colunas.
Sempre o primeiro número é a linha e o segundo é a coluna.

Para declarar matrizes e inicializar, devemos colocar cada linha entre chaves {}, e separar elas por vírgulas, veja:
float notas[2][5] = { {8.0, 7.5, 8.5, 9.0, 8.0 }, {8.9, 9.0, 8.6, 8.4, 8.0 } };

Uma maneira mais simples de ver essas linhas e colunas, como tabela, é da seguinte maneira:
float notas[2][5] = { {8.0, 7.5, 8.5, 9.0, 8.0 },
{8.9, 9.0, 8.6, 8.4, 8.0 } };

Para declarar a matriz 3x5, fazemos::
float notas[3][5];

Veja como fica nossa matriz, ou tabela, declarada e inicializada:
float notas[3][5] = { {8.0, 7.5, 8.5, 9.0, 8.0 }, {8.9, 9.0, 8.6, 8.4, 8.0 }, {6.8, 7.1, 7.0, 7.6, 6.5 } };

De uma maneira mais fácil de entender:
float notas[3][5] = { {8.0, 7.5, 8.5, 9.0, 8.0 },
{8.9, 9.0, 8.6, 8.4, 8.0 },
{6.8, 7.1, 7.0, 7.6, 6.5 } };

Note que notas[0] se refere ao vetor de notas de Matemática.
Note que notas[1] se refere ao vetor de notas de Física.
Note que notas[2] se refere ao vetor de notas de Química.

Por exemplo: qual foi a quarta nota de Física do aluno?
Ora, o vetor de Física é notas[1], e a quarta nota é o elemento [3] desse vetor.
Então a quarta nota de Física do aluno está armazenada em: notas[1][3], que é 8.4

Generalizando, para declarar uma matriz de ‘linha’ linhas e de ‘coluna’ colunas, fazemos:
tipo nome[linha][coluna];

Para acessar o elemento da i-ésima linha e de j-ésima coluna, acessamos pela variável:
nome[i][j];

É uma variável como outra qualquer em linguagem de programação C. Podemos somar, incrementar, zerar etc.

Bom, agora a apostila C Progressivo sai da teoria para, como de costume, ir para a prática com exemplos resolvidos e comentados.

- **Exemplo de código C: Criar e exibir uma matriz 3x3**

Crie um aplicativo em C que peça ao usuário para preencher uma matriz 3x3 com valores inteiros e depois exiba essa matriz.

A grande novidade, e importância, nesse tipo de aplicativo são os laços for aninhados, ou seja, um dentro do outro, e do uso do *#define*, para tratar da constante DIM, que representa a dimensão da matriz.
É importante, como foi explicado no artigo passado, você usar *#define* ou *const* para trabalhar com constantes em C.

Primeiro criamos um laço que vai percorrer todas as linhas da matriz. Podemos, e devemos, ver cada linha como um vetor de 3 elementos.

Dentro de cada linha, temos que percorrer cada elemento do do vetor e fornecer seu valor. Fazemos isso através de outro laço for, que ficará responsável pelas 'colunas', formando nossos laços aninhados.

Para imprimir, o esquema é exatamente o mesmo. Imprimimos linha por linha, e em cada linha, imprimimos coluna por coluna.

```
#include <stdio.h>
#define DIM 3
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito!
// Artigos, apostilas, tutoriais e
```

```c
// vídeo-aulas sobre a linguagem de programação C!

int main()
{
    int matriz[DIM][DIM];
    int linha, coluna;

    //escrevendo na Matriz
    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
        {
            printf("Elemento [%d][%d]: ", linha+1, coluna+1);
            scanf("%d", &matriz[linha][coluna]);
        }

    // imprimindo a matriz na tela
    for(linha = 0 ; linha < DIM ; linha++)
    {
        for(coluna = 0 ; coluna < DIM ; coluna++)
            printf("%3d", matriz[linha][coluna]);

        printf("\n"); //após cada linha ser impressa, um salto de linha
    }
}
```

- **Exemplo 2: Como calcular o traço de uma matriz em C**

Use o programa feito no exemplo anterior para calcular o traço de uma matriz.
Lembrando que o traço de uma matriz é a soma dos elementos da diagonal principal.

Os elementos das diagonal principal são os que tem índice da linha igual ao índice da coluna:
matriz[0][0], matriz[1][1] e matriz[2][2].

Aproveitamos o laço for das linhas para calcular a soma desses elementos:
matriz[linha][linha]

Visto que a variável 'linha', assim como a 'coluna', vão de 0 até 2.
Logo, nosso código C fica:

```c
#include <stdio.h>
#define DIM 3
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C, online e gratuito
// Artigos, apostilas, tutoriais e
// vídeo-aulas sobre a linguagem de programação C

int main()
{
    int matriz[DIM][DIM];
    int linha, coluna, traco = 0;

    //escrevendo na Matriz
    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
        {
            printf("Elemento [%d][%d]: ", linha+1, coluna+1);
            scanf("%d", &matriz[linha][coluna]);
        }

    // imprimindo a matriz na tela
    for(linha = 0 ; linha < DIM ; linha++)
    {
        for(coluna = 0 ; coluna < DIM ; coluna++)
            printf("%3d", matriz[linha][coluna]);

        traco += matriz[linha][linha];
        printf("\n"); //após cada linha ser impressa, um salto de linha
    }

    printf("\nTraco da matriz: %d\n", traco);
}
```

Como passar vetores, ou arrays, e Matrizes (vetores multidimensionais) para funções em C

Agora que você já domina bem o uso das funções e dos vetores em C, vamos unir esses dois conhecimentos e ensinar você como passar vetores (unidimensionais, multidimensionais, matriz), ou arrays, para funções.

Como o curso C Progressivo vai mostrar, para programar em C é bem comum passar vetores, e outras estruturas de dados, para funções.

Como você viu nos exemplos passados de nossa apostila de C, dá um certo trabalhinho lidar com vetores.
E para o código não ficar confuso e grande na *main()*, costumamos colocar os códigos que operam vetores em funções.

- **Como passar um vetor parar uma função em C**

Embora tenhamos batido na tecla sobre o fato de vetores serem um conjunto de variáveis, como outras quaisquer, existem alguns detalhes que precisamos saber na hora de passar os vetores para as funções, principalmente no cabeçalho de declaração da função.

Mas não precisa entrar em desespero, pois os detalhes são para facilitar nossa vida.

No caso de um vetor unidimensional (vetor comum, de uma dimensão), precisamos colocar apenas o par de colchetes – [ ] -após o nome da variável, e vazio.
Isso mesmo, vazio. Não importa se seu vetor tem 1, 10 ou 1 milhão de elementos.
As funções em C apenas precisam saber qual o tipo da variável e se é um vetor.

Então, a sintaxe de uma função que recebe um vetor é:
retorno nomeDaFuncao(tipo vetor[] , ...)

Por exemplo, uma função que recebe um vetor de inteiros e retorna um inteiro:
int funcao(int numeros[])

Uma função sem retorno que recebe um vetor/array de floats:
void funcao2(float decimais[])

O outro detalhe vem na hora de invocar a função e passar o vetor como argumento.
Para fazer isso basta colocar o nome do vetor, não precisa dos pares de colchetes [] nem da dimensão do vetor.

Por exemplo, para passar vetores as funções que demos como exemplo:
funcao(numeros);
funcao2(numeros):

Exemplo: Programa que calcula média aritmética, com vetor e função
Crie um programa em C que peça 5 números ao usuário, armazene esses dados em um vetor, passe esse vetor para uma função que retorna o valor da média dos números desse vetor.

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

float media(float numeros[], int n)
{
    int count;
    float media=0.0;

    for(count=0 ; count<n ; count++)
        media += numeros[count];

    return (media/n);
}

int main(void)
{ float numeros[5]; int count; for(count=0 ; count < 5 ; count++) {
printf("Entre com o numero %d: ", count+1); scanf("%f", &numeros[count]); }
printf("A media desses numeros e: %.2f\n", media(numeros, 5));
    return 0;
}
```

- **Como passar matrizes (vetores multidimensionais) para funções**

Este tópico está separado do anterior, pois aqui temos mais um detalhe que precisamos aprender sobre a passagem de matrizes para funções.

Sabemos que matrizes multidimensionais são aqueles que possuem duas ou mais dimensões.
Vimos também em nosso tutorial sobre matrizes (ou vetoresmultidimensionais) em C, que cada dimensão da matriz exige um paz de colchetes [ ].

Pois bem, o detalhe para passar matriz para as funções é que o primeiro par de colchetes pode sempre ir vazio, e os demais preenchidos.

Por exemplo, vamos supor que você criou um jogo da velha, que é uma matriz 3x3, e uma função que checa se algum jogador ganhou o jogo completando alguma linha.

Essa função teria o seguinte cabeçalho:
int checaLinha(int matriz[][3])

Ou seja, não precisamos especificar o número de linhas que a matriz tem, mas somente o número de colunas.

Exemplo de código: Como preencher, exibir uma matriz e seu traço, passando a matriz para uma função em C

Crie um programa em C que peça para o usuário preencher uma matriz 3x3, que exiba ela e o valor de seu traço em seguida. Use funções para preencher a matriz, exibir e calcular o traço.

Esse aplicativo em C já foi totalmente resolvido como exemplos, no artigo passado de nossa apostila de C.

O que vamos fazer aqui é criar três funções: uma para preencher a matriz, outra pra exibir e uma que calcula o traço.

Note como a nossa função *main()* ficou bem limpa.
Esse é o ideal, é padrão C Progressivo!

```c
include <stdio.h>
#define DIM 3
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C, Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

void preencher(int matriz[][DIM])
{
    int linha,
        coluna;
    for(linha=0 ; linha < DIM ; linha++)
        for(coluna=0 ; coluna < DIM ; coluna++)
        {
            printf("Entre com o elemento matriz[%d][%d]: ", linha+1, coluna+1);
            scanf("%d", &matriz[linha][coluna]);
        }
}
void exibir(int matriz[][DIM])
{
    int linha,
        coluna;
    for(linha=0 ; linha < DIM ; linha++)
    {
        for(coluna=0 ; coluna < DIM ; coluna++)
            printf("%3d ", matriz[linha][coluna]);

        printf("\n");
    }
}

int traco(int matriz[][DIM])
{
    int count,
        traco=0;
    for(count=0 ; count < DIM ; count++)
        traco += matriz[count][count];
    return traco;
}
int main(void)
{ int matriz[DIM][DIM]; preencher(matriz); exibir(matriz); printf("\nTraco da
matriz: %d\n", traco(matriz));

    return 0;
}
```

Passagem por Referência - Como copiar vetor e matriz em C

Se você tiver notado bem nosso tutorial passado, sobre como passar vetores(arrays) e matrizes(vetores multidimensionais) em linguagem C, nós passamos vetores para funções, e essas alteravam os valores contidos nesses vetores.

Em um tutorial da seção sobre Funções em C, sobre variáveis locais, também dissemos que quando passamos variáveis para funções, seus valores não são alterados, pois as funções trabalhavam em cima de uma cópia da variável, e não na variável em sim, e dissemos que isso era chamado de passagem por valor.

Agora, em nossa apostila de C, vamos aprender um pouco sobre passagem por referência, em C, que é um tipo especial de passagem.

- **Vetor e Matriz: Passagem por referência**

A passagem de vetores e matrizes para funções é feita por referência. Ou seja, mandamos uma espécie de referência para as funções, em vez de mandar o próprio vetor ou uma cópia dele, como ocorre na passagem por valor.

- **O que é uma referência, em linguagem C?**

É um endereço. Sim, um endereço na memória.
Lembra que falamos que, ao declarar variáveis, o C aloca um espaço em memória? Então, todo espaço da memória é identificado por um endereço.

É esse endereço que é passado para as funções, quando passamos vetores para funções.
As funções, quando recebem um vetor como argumento, passa a trabalhar no endereço que lhe foi passado. E o que tem nesse endereço?
Ora, o vetor que declaramos.

Logo, quando a função trabalhar com vetor, trabalha com o vetor real. Por isso os valores de um vetor podem ser alterados dentro de uma função.

- **Como copiar vetores e matrizes em C**

Muitas vezes não queremos alterar nosso vetor.
Em muitos programas, precisamos extrair informação dos vetores, fazendo cálculos e modificações no mesmo, mas não queremos que esse vetor seja alterado.

O que fazer, então?
Copiar esse vetor/array/matriz e trabalhar com a cópia.
Porém, diferente das variáveis comuns, não podemos simplesmente igualar um vetor a outro.

Então, **para copiar um vetor/array/matriz em Linguagem C, precisamos copiar elemento por elemento**.

Vamos mostrar através de um exemplo como as funções realmente alteram os valores dos vetores, e como copiar um vetor.

Exemplo: Como copiar um vetor em C
Crie um programa que peça para o usuário 5 números, faça uma cópia desse idêntica desse vetor, outra cópia mas com os valores dobrados dos elementos. Por fim, mostre todos os 3 vetores: original, copiado e dobrado. Use funções.

Esse exemplo mostra bem como as funções alteram os valores dos vetores. Primeiro criamos a função recebe, que irá receber os dados do usuário, que serão armazenados no vetor ‘original’.

Depois fazemos a função copiar, que recebe o vetor original e o vetor que vamos usar pra fazer a cópia. Note que temos que fazer a cópia elemento por elemento.

A função dobrar é bem parecida com a função que copia. A diferença é que a cópia é dobrada.
Como é necessário copiar elemento por elemento, para dobrar os valores de um vetor, é necessário multiplicar por 2 cada elemento também.

E por fim, uma função já velha conhecida nossa, que mostra os elementos de um vetor.

```c
#include <stdio.h>
#define DIM 5
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

void recebe(int original[])
{
    int count = 0;

    for(count = 0; count < DIM ; count++)
    {
        printf("Entre com o elemento %d: ", count+1);
        scanf("%d", &original[count]);
    }
}

void copiar(int copiado[], int original[])
{
    int count;

    for(count=0 ; count < DIM ; count++)
        copiado[count] = original[count];
}

void dobrar(int dobrado[], int original[])
{
    int count;

    for(count=0 ; count < DIM ; count++)
        dobrado[count] = 2 * original[count];
}

void exibir(int vet[])
{
    int count;

    for(count=0 ; count < DIM ; count++)
        printf("%3d ", vet[count]);
}

int main(void)
{ int original[DIM], copia[DIM], dobrado[DIM]; recebe(original); copiar(copia,
original); dobrar(dobrado, original); printf("Vetor original: "); exibir(original);
```

```
printf("\nVetor copiado : "); exibir(copia); printf("\nVetor dobrado : ");
exibir(dobrado); return 0; }
```

Exemplo de código: Como copiar uma matriz em linguagem C

Faça o exemplo passado, mas agora o usuário vai inserir uma matriz 3x3.

Veja como se copia uma matriz em C:

```
#include <stdio.h>
#define DIM 3
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

void recebe(int original[][DIM])
{
    int linha,
        coluna;

    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
        {
            printf("Digite o elemento [%d][%d]: ", linha+1, coluna+1);
            scanf("%d", &original[linha][coluna]);
        }
}

void copiar(int copiado[][DIM], int original[][DIM])
{
    int linha,
        coluna;

    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
            copiado[linha][coluna] = original[linha][coluna];
}

void dobrar(int dobrado[][DIM], int original[][DIM])
{
    int linha,
        coluna;

    for(linha = 0 ; linha < DIM ; linha++)
```

```
        for(coluna = 0 ; coluna < DIM ; coluna++)
            dobrado[linha][coluna] = 2 * original[linha][coluna];
}

void exibir(int matriz[][DIM])
{
    int linha,
        coluna;

    for(linha = 0 ; linha < DIM ; linha++)
    {
        for(coluna = 0 ; coluna < DIM ; coluna++)
            printf("%3d ", matriz[linha][coluna]);

        printf("\n");
    }
}

int main(void)
{
   int original[DIM][DIM], copia[DIM][DIM], dobrado[DIM][DIM];
   recebe(original);
   copiar(copia, original);
   dobrar(dobrado, original);
   printf("Vetor original: \n");
   exibir(original);
   printf("\nVetor copiado : \n");
   exibir(copia);
    printf("\nVetor dobrado : \n");

    exibir(dobrado);
    return 0;
}
```

Exercícios sobre Vetores e Matrizes em C

Agora que já estudou tudo sobre os vetores (também conhecido por arrays) e matrizes (também conhecidas como vetores multidimensionais) em linguagem C, está na hora de treinar nossos conhecimentos.

Embora exista várias maneiras de resolver os exercícios a seguir, tente resolver usando vetores.

Aproveite para testar a boa prática de programação: divida as funções de modo que cada uma delas faça uma coisa específica. Use sempre funções, deixe sua *main()* a mais limpa possível.

Exercícios sobre vetores (arrays) e Matriz em C

00. Crie um programa em C que peça 10 números, armazene eles em um vetor e diga qual elemento é o maior, e seu valor.

01. Crie um programa em C que peça 10 números, armazene eles em um vetor e diga qual elemento é o menor, e seu valor.

02. Crie um programa em C que peça 10 números, armazene eles em um vetor e diga qual elemento é o maior, qual é o menor e que seus valores.

03. Crie um aplicativo em C que peça um número inicial ao usuário, uma razão e calcule os termos de uma P.A (Progressão Aritmética), armazenando esses valores em um vetor de tamanho 10.

04. Crie um aplicativo em C que peça um número inicial ao usuário, uma razão e calcule os termos de uma P.G (Progressão Geométrica), armazenando esses valores em um vetor de tamanho 10.

05. Escreve um programa que sorteio, aleatoriamente, 10 números e armazene estes em um vetor.
Em seguida, o usuário digita um número e seu programa em C deve acusar se o número digitado está no vetor ou não. Se estiver, diga a posição que está.

Desafio 0: Criando um tabuleiro de Jogo da Velha
Crie um tabuleiro de jogo da velha, usando uma matrizes de caracteres (char) 3x3, onde o usuário pede o número da linha (1 até 3) e o da coluna (1 até 3). A cada vez que o usuário entrar com esses dados, colocar um 'X' ou 'O' no local selecionado.

Desafio 1: Como criar um programa que checa se o número é palíndromo
Número palíndromo é aquele que, se lido de trás para frente e de frete para trás, é o mesmo.
Exemplos: 2112, 666, 2442 etc

Dica: extraia os dígitos, coloque cada dígito do número em um vetor e faça a comparação dos números (primeiro com o último, segundo com o penúltimo, terceiro com o anti-penúltimo etc).

O número deve ter 4 dígitos.

Jogo da Velha em C

Agora que já ensinamos os conceitos básicos da linguagem C, os testes condicionais e loopings, Funções e Vetores (matrizes), vamos colocar em prática nossos conhecimentos para criar algo interessante:

- **O Famoso Jogo da Velha**

Nesse tutorial vamos ensinar as regras como se jogar em nosso aplicativo, bem como a lógica do jogo.
É importante você tentar fazer o seu, pois só assim você irá aprender.

Não tenha medo, a primeira vez que tentamos, o código fica gigantesco e confuso.
Mas o começo é assim mesmo.

No próximo artigo, vamos comentar cada parte do código aqui mostrado, ensinando passo-a-passo como se cria o jogo.

- **Como jogar o Jogo da Velha**

O jogo da velha é um jogo para duas pessoas.
A estrutura do jogo é apenas um tabuleiro 3x3 posições, onde os jogadores marcam ou 'X' ou 'O' em cada campo do tabuleiro.

Para vencer o jogo, um dos jogadores deve preencher uma linha, uma coluna ou uma diagonal inteira com 'X' ou 'O'. O primeiro que fizer isso vence, e o jogo termina.
Caso o tabuleiro fique totalmente preenchido e ninguém ganhar, teremos um empate.

Nessa versão de nosso jogo, o primeiro jogador será representado pelo 'X' e o segundo pelo 'O'.

Como nosso aplicativo roda na tela de comandos (DOS ou Terminal), a cada jogada precisamos fornecer dois números que representam a posição do tabuleiro em que queremos jogar: a linha e a coluna, nessa ordem.
As linhas, bem como as colunas, variam de 1 até 3 e somente os locais vazios podem ser escolhidos como jogada.

Lembrando que a linha é horizontal e coluna é vertical.

- **Recomendações**

Com base nas regras acima, tente criar seu próprio jogo da velha.
Essa tentativa provavelmente vai ser frustrada. Não deixe se abater se não souber nem por onde começar.

Como recomendamos várias vezes ao longo do curso, faça por partes.
Tente exibir na tela o tabuleiro, atrás de uma matriz 3x3.
Peça ao usuário o número da linha e da coluna.
Lembre-se: a cada jogada, é um dos dois jogadores que irá jogar, sempre alternando.
Faça com que o primeiro marque um 'X' na tela e o segundo um 'O'.

Evite usar números mágicos, use #define DIM.
Deixe sua função *main()* a mais enxuta e limpa possível.
Divida ao máximo seu programa em funções, onde cada função deve fazer uma tarefa de cada vez.

Evite usar *fflush()* ou *system()*.
Para evitar *fflush()*, você pode usar as funções getchar() ou trabalhar com números.
Em vez de usar *system("cls")* caso use Windows, ou *system("clear")* caso use Linux, simplesmente imprima na tela vários caracteres de *newline*.
Pois, como já havíamos dito, essas funções são lentas e há problemas de segurança ao usar elas.

Embora nosso curso seja focado nos iniciantes, é importante você adquirir esses hábitos desde o início, para eliminar os vícios.

Uma outra maneira de você treinar, é vendo o tutorial comentado do código do Jogo da Velha em C, sem olhar pro código, e tentar criar a partir do que você entendeu da lógica do programa, que vem logo a seguir na apostila.

Código completo do Jogo da Velha em C

```c
#include <stdio.h>
#define DIM 3
int vez;

int menu(void);
void clear(void);
void zeraTabuleiro(int tabuleiro[][DIM]);
void exibeTabuleiro(int tabuleiro[][DIM]);
void jogar(int tabuleiro[][DIM]);
int checaLocal(int tabuleiro[][DIM], int linha, int coluna);
int checaLinha(int tabuleiro[][DIM]);
int checaColuna(int tabuleiro[][DIM]);
int checaDiagonal(int tabuleiro[][DIM]);
int checaEmpate(int tabuleiro[][DIM]);
int checaTermino(int tabuleiro[][DIM], int vez);
void jogada(int tabuleiro[][DIM]);

int main(void)
{
    int tabuleiro[DIM][DIM],
        continuar;

    do
    {
        vez=1;
        continuar = menu();
        if(continuar == 1)
            jogar(tabuleiro);

    }while(continuar);

    return 0;
}

int menu(void)
{
```

```c
    int opcao;

        printf("\t\tJogo da Velha 1.0 - Site C Progressivo\n");
        printf("\n1.Jogar\n");
        printf("0.Sair\n");
        printf("\nOpcao: ");

        scanf("%d", &opcao);

        switch(opcao)
        {
           case 1:
           case 0:
                break;
           default:
                clear();
                printf("Opcao invalida. Tente de novo!\n");
        }

    return opcao;
}

void clear(void)
{
    int count=0;

    while(count != 100)
    {
        putchar('\n');
        count++;
    }
}

void zeraTabuleiro(int tabuleiro[][DIM])
{
    int linha, coluna;
    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
            tabuleiro[linha][coluna] = 0;
}

void exibeTabuleiro(int tabuleiro[][DIM])
{
    int linha, coluna;
    putchar('\n');
```

```
    for(linha = 0 ; linha < DIM ; linha++)
    {
        for(coluna = 0 ; coluna < DIM ; coluna++)
        {
            if(tabuleiro[linha][coluna] == 0)
                printf("    ");
            else
                if(tabuleiro[linha][coluna] == 1)
                    printf("  X ");
                else
                    printf("  O ");

            if(coluna != (DIM-1))
                printf("|");
        }
        putchar('\n');
    }
    putchar('\n');
}

void jogar(int tabuleiro[][DIM])
{
    int continua;
    zeraTabuleiro(tabuleiro);

    do
    {
        clear();
        exibeTabuleiro(tabuleiro);
        jogada(tabuleiro);

    }while(checaTermino(tabuleiro, vez) != 1);
}


int checaLocal(int tabuleiro[][DIM], int linha, int coluna)
{
    if(linha < 0 || linha > (DIM-1) || coluna < 0 || coluna > (DIM-1) ||
tabuleiro[linha][coluna] != 0)
        return 0;
    else
        return 1;
}
```

```
int checaLinha(int tabuleiro[][DIM])
{
    int linha, coluna,
        soma;

    for(linha = 0 ; linha < DIM ; linha++)
    {
        soma=0;

        for(coluna = 0 ; coluna < DIM ; coluna++)
            soma += tabuleiro[linha][coluna];

        if(soma==DIM || soma == (-1)*DIM)
            return 1;
    }

    return 0;
}

int checaColuna(int tabuleiro[][DIM])
{
    int linha, coluna,
        soma;


    for(coluna = 0 ; coluna < DIM ; coluna++)
    {
        soma=0;

        for(linha = 0 ; linha < DIM ; linha++)
            soma += tabuleiro[linha][coluna];

        if(soma==DIM || soma == (-1)*DIM)
            return 1;
    }

    return 0;
}

int checaDiagonal(int tabuleiro[][DIM])
{
    int linha,
        diagonal_principal=0,
        diagonal_secundaria=0;
```

```
    for(linha = 0 ; linha < DIM ; linha++)
    {
        diagonal_principal += tabuleiro[linha][linha];
        diagonal_secundaria += tabuleiro[linha][DIM-linha-1];
    }

    if(diagonal_principal==DIM || diagonal_principal==(-1)*DIM ||
      diagonal_secundaria==DIM || diagonal_secundaria==(-1)*DIM)
      return 1;

    return 0;
}

int checaEmpate(int tabuleiro[][DIM])
{
    int linha, coluna;

    for(linha = 0 ; linha < DIM ; linha++)
        for(coluna = 0 ; coluna < DIM ; coluna++)
            if(tabuleiro[linha][coluna] == 0)
                return 0;

    return 1;
}

int checaTermino(int tabuleiro[][DIM], int vez)
{
    if(checaLinha(tabuleiro))
    {
        printf("Jogo encerrado. Jogador %d venceu !\n\n", (vez%2)+1);
        exibeTabuleiro(tabuleiro);
        return 1;
    }

    if(checaColuna(tabuleiro))
    {
        printf("Jogo encerrado. Jogador %d venceu !\n\n", (vez%2)+1);
        exibeTabuleiro(tabuleiro);
        return 1;
    }

    if(checaDiagonal(tabuleiro))
    {
        printf("Jogo encerrado. Jogador %d venceu !\n\n", (vez%2)+1);
        exibeTabuleiro(tabuleiro);
```

```
        return 1;
    }

    if(checaEmpate(tabuleiro))
    {
        printf("Jogo encerrado. Ocorreu um empate! !\n\n");
        exibeTabuleiro(tabuleiro);
        return 1;
    }

    return 0;
}

void jogada(int tabuleiro[][DIM])
{
    int linha, coluna;
    vez++;
    printf("\n--> Jogador %d \n", (vez % 2) + 1);

    do
    {
        printf("Linha: ");
        scanf("%d", &linha);
        linha--;

        printf("Coluna: ");
        scanf("%d", &coluna);
        coluna--;

        if(checaLocal(tabuleiro, linha, coluna) == 0)
            printf("Posicao ocupada ou inexistente, escolha outra.\n");

    } while(checaLocal(tabuleiro, linha, coluna) == 0);

    if(vez%2)
        tabuleiro[linha][coluna] = -1;
    else
        tabuleiro[linha][coluna] = 1;
}
```

- ## Lógica do Jogo da Velha em C

Não existe apenas uma maneira de se criar um programa em C, qualquer que seja ela.
Isso vai depender de vários fatores: criatividade, tempo, conhecimento, eficiência, clareza do código entre outros detalhes.

Portanto, a lógica aqui apresentada é apenas uma, das infinitas que existem e que você encontrar por aí.

Aqui, vamos criar o tabuleiro do jogo com uma matriz 3x3 de números inteiros.
Quando uma casa do tabuleiro está vazia, ela contém o número 0.
Quando o primeiro jogador marca um local, ele está colocando o número 1 nessa casa.
E quando é o segundo, ele está colocando o número -1 nesse local do tabuleiro.

O jogo termina quando uma linha, coluna ou diagonal é completada por números 1 ou -1.
Para saber se a linha está completa, basta verificar a soma. Se ela for 3 ou -3, é porque alguém ganhou.
Se todos os campos do tabuleiro forem preenchidos e ninguém somou 3 ou -3 em uma fila, houve empate.

Uma variável inteira chamada ‘vez’ é a que controlará se a vez do primeiro ou do segundo jogador.
Fazendo o resto da divisão por 2, e incrementando essa variável a cada rodada, teremos sempre os números 0 e 1 alternadamente, para saber de quem é a vez de jogar.

- ## As funções do Jogo da Velha em C

A função ***main()*** cria o tabuleiro 3x3 e um inteiro ‘continuar’, que será usado no laço do while que irá controlar se o jogador vai continuar jogando ou não.
Essa variável recebe o resultado da função ***menu()***, que exibe para o usuário a opção de Jogar ou Sair.
Se o jogador escolher 1, é porquê quer jogar.
Se escolher 0, é porquê quer sair.
Se escolher qualquer outro número, digitou errado.

Essa função retorna o número da opção para a função ***main()***.
De volta na main, se a opção recebida for 1, ‘continuar’ tem valor lógico TRUE e o jogo prossegue. Se a opção recebida for 0, ‘continuar’ tem valor lógico FALSE e o programa termina. Para qualquer outro valor, o laço do while continua a se repetir, até o indivíduo entrar com uma opção válida.

Note que você pode incrementar esse menu, deixando sempre 0 para sair do jogo.
Pode alterar para jogar Humano x Máquina, escolher quem vai ser ‘X’ e quem vai ser ‘O’, quem começa e por ai vai.

Na ***main()***, se a opção recebida for 1, chamamos a função ***jogar()*** passando o tabuleiro como argumento. Essa função que ficará responsável pelo desenrolar do jogo.
Na função ***jogar()***, a primeira coisa a ser feita é colocar 0 em todas as posições do tabuleiro, pois pode ter lixo lá, e fazemos isso chamando a função ***zeraTabuleiro()*** e passando a matriz para ele zerar. Como matriz é um vetor, a passagem é feita automaticamente por referência, então a matriz tem seus valores alterados, todos para 0.

Voltando a função ***jogar()***, ela cria um looping com o laço do while, que se repete enquanto o jogo não termina. E cada iteração desse laço é uma jogada.

Porém, antes da jogada, usamos a função ***clear()***, que simplesmente imprime vários caracteres de *newline* ***\n*** na tela.

Você pode até usar o comando ***system(“cls”)*** para Windows ou ***system(“clear”)*** para Linux, mas são funções inseguras e pesadas, que consomem muito processamento das máquinas, mas nada que você vá notar em um programa simples como este, porém é bom cultivarmos boas práticas de programação desde cedo, e o efeito de imprimir várias quebras de linha não muda em nada a estética do jogo, só deixa sua aplicação mais robusta e segura.

Após limpar a tela, vamos exibir o tabuleiro.
Porém, ficaria muito feio e incômodo pro usuário um tabuleiro cheio de números 0, 1 e -1.

Ao invés disso, vamos percorrer todos os números da matriz inteiros e onde encontrarmos o número 0, vamos imprimir um espaço vazio. Onde tiver número 1, imprimimos o ‘X’ e se nos depararmos com o número -1, exibimos ‘O’ pro usuário, naquela posição.

Colocamos colunas | após cada coluna (exceto após a última coluna).
Isso tudo é feito na função ***exibeTabuleiro()***.

A jogada é feita através da função ***jogada()***, que pede ao usuário uma linha e coluna para marcar no tabuleiro.

Note que o usuário vai colocar números entre 1 e 3, mas seu tabuleiro trabalha com índices entre 0 e 2. Ou seja, quando o usuário insere o número da linha e o da coluna, devemos decrementar em 1 cada um desses valores. Após isso, usamos a função ***checaLocal()***, que recebe o tabuleiro, a linha e a coluna que o usuário inseriu e vai no tabuleiro ver se aquela posição está vazia, ou seja, se tem o número 0 naquela casa. Se não estiver, o laço do while continua a se repetir até o usuário inserir valores de linha e coluna que representem uma casa vazia no tabuleiro.

E além de checar se está vazia, temos que checar se o usuário inseriu valores corretos de linha e coluna (não pode ser menor que 1 nem maior que 3).

Quando entramos nessa função ***jogada()***, a variável global ‘vez’ é incrementada em 1, e fazemos o resto da divisão por 2, que vai ter resultados 0 e 1 alternadamente. Somamos 1 a esses resultados, e o usuário vê se é a vez do jogador 1 ou jogador 2 de jogar.

Após ter escolhido o local e checado que ele está vazio, vamos marcar aquele lugar no tabuleiro, usando a variável ‘vez’. Se o resto da divisão por 2 for 0, é porque é o jogador 1 que vai jogar e marcamos o número ‘1’ na casa, mas se o resto da divisão for 1, é porque é o jogador 2 que vai jogar e marcamos o número ‘-1’ no local.

Esse procedimento de jogada vai se repetir enquanto o jogo não termina.
E quem diz se o jogo termina ou não, é a função ***checaTermino()***, que nada mais é que um conjunto de outras funções dentro dela, como:

checaLinha() – essa função checa a soma de cada linha do tabuleiro, se alguma for 3 ou -3, é porque alguém ganhou.
checaColuna() – faz a mesma coisa da função anterior, mas checa as colunas
checaDiagonal() – essa função checa as duas diagonais de nosso tabuleiro, pra ver se a soma de uma delas é 3 ou -3

Se alguma das funções anteriores acusar soma 3 ou -3, elas retornam valor 1 (TRUE), então o teste condiciona *if* é acionado, a mensagem de vitória é dada e a função retorna valor 1, dizendo que o jogo terminou.

Temos ainda a função ***checaEmpate()***, que checa se ainda existe alguma casa livre no tabuleiro. Ou seja, ela sai em busca do número 0, que representa casa vazia, se encontrar retorna valor lógico FALSE, indicando que não foi empate.
Porém, se não encontrar 0, é porque o tabuleiro está preenchido e ela retorna valor 1 e a mensagem de empate.

Após terminar o jogo, com alguém vencendo ou dando empate, voltamos pro looping da função ***main()***, que pergunta novamente se você quer jogar ou não.
Se sim, tudo ocorre de novo.
Se não, o programa em C termina.

Ponteiros

É fato: ponteiros, ou apontadores, é considerado o assunto mais difícil na linguagem C.
Isso se devo ao fato de ser uma ideia nova, uma abstração diferente e até então nunca vista pelo programador.

Mas todos passaram por essa dificuldade inicial. E aprendendo bem ponteiros, você terá meio caminho andado rumo ao sucesso profissional como um programador C.

Aprender, entender e manipular BEM ponteiros é um diferencial para o programador.
É um assunto importante, que servirá de base para nosso estudo de *strings*, alocação dinâmica de memória e estrutura de dados (listas, pilhas e filas).

Na verdade, ponteiros não é um assunto difícil nem complexo, é apenas um assunto que irá precisar de sua atenção e estudo. E o curso gratuito e online C Progressivovai te ajudar a entender e dominar completamente o uso e conceito dos ponteiros / apontadores.

Introdução ao uso dos ponteiros - Endereços de memória

Desde o início de nosso curso de C, usamos os mais diversos tipos de dados, como *int* para números inteiros, *float* e *double* para números decimais, *char* para caracteres etc.

Vamos agora apresentar um novo tipo dado: os ponteiros.

- **O que são ponteiros em C ?**

Não se assuste com o tanto de tipos de variáveis, elas são feitas, em sua maioria, para que os humanos possam entender e trabalhar mais facilmente com os diversos tipos de dados.

Porém, para o computador, não existe praticamente diferença alguma entre as variáveis, para ele é tudo bit, é tudo 1 ou 0.
Um meio bastante usado nos hardwares para administrar esse número gigantesco de 1's e 0's, é através do endereçamento.

Cada trecho da memória tem um endereço único. Não existem dois bits, em uma máquina, que tenha o mesmo endereço de memória.
O ato de selecionar, ou alocar, um espaço de memória em C é feito no momento da declaração.

Endereços de memória são um tipo de dado tão importante, ou até mais, que qualquer outro tipo de dado. Quem já trabalhou com eletrônica, em baixo nível, sabe que endereçamento é uma parte essencial de qualquer circuito digital. Nossos computadores possuem diversos dispositivos que são responsáveis somente pelo endereçamento.

E é isso que o um ponteiro é: um tipo de dado que serve para indicar, ou armazenar, um endereço de memória.

Um ponteiro não é um inteiro, é um tipo que armazena o endereço em que o inteiro está alocado.
Um ponteiro não é um float ou double, ponteiro é um tipo de dado que armazena o endereço em que o float ou double está.
Um ponteiro não é um char, ponteiro é um tipo de dado que pode armazenar o endereço em que um caractere está.

É muito importante que você entenda, e se lembre bem disso, pois é muito comum os iniciantes confundirem ponteiros com outros tipos de dados.

Essa confusão é uma dificuldade natural que todas as pessoas têm ao se depararem pela primeira vez com o uso dos ponteiros.
O interessante é que as pessoas não confundem inteiro com um caractere, ou caractere com um número decimal, mas confundem ponteiro com números inteiros.

Isso se deve ao fato dos ponteiros serem um tipo de abstração, criado especialmente para facilitar o trabalho da computação em baixo nível, da computação que mexe diretamente com a memória de seu computador, poder este que pouquíssimas linguagens possuem.

- **Como saber o endereço de memória de uma variável: &**

Sempre que declaramos uma variável e usamos ela, estamos trabalhando com seu valor.

Por exemplo:

```
numero1 = 1;
numero2 = 2;
letra1 = ‘a’;
letra2 = ‘b’;
```

O valor da variável ‘numero1’ é 1, o valor da variável ‘numero2’ é 2, o valor da variável ‘letra1’ é ‘a’ e o valor da variável ‘letra2’ é ‘b’.

Fixe bem esse detalhe: esse é o valor que está armazenado na memória, essas variáveis são um conjunto de bits, um conjunto de informações, um valor.

Agora vamos descobrir em qual posição da memória esses valores estão.
Para isso, basta colocarmos o símbolo de E comercial antes da variável: &

Para saber o endereço da variável ‘numero1’, fazemos: &numero1
Para saber o endereço da variável ‘numero2’, fazemos: &numero2
Para saber o endereço da variável ‘letra1’, fazemos: &letra1

Para saber o endereço da variável ‘letra2’, fazemos: &letra2

Para facilitar a visualização do usuário, podemos imaginar a memória como um vetor gigantesco de espaços, e esses espaços são numerados com números inteiros.

Veja bem, embora seja um inteiro, não quer dizer que o valor seja inteiro. Todos os endereços são números inteiros, mas nem todo o valor armazenado dentro daquele endereço de memória é inteiro.

Vamos fazer um exemplo para entender melhor a diferença entre valor e endereço de uma memória.

- **Exemplo de código: Vendo o valor e endereço de uma variável**

Crie um programa em C que declara dois números inteiros e dois caracteres do tipo char (todos devidamente inicializados).
Em seguida, mostre o VALOR de cada variável, bem como seu ENDEREÇO
Depois, altere os valores das variáveis e mostre novamente o VALOR e ENDEREÇO de cada variável desta.

Após rodar esse exemplo, você verá a clara diferença entre o VALOR e o ENDEREÇO de uma variável na memória de seu computador.
O valor é aquela informação que você inicia, e endereço é um número inteiro ENORME.

O valor é aquela informação que é alterada, já o endereço de uma variável permanece CONSTANTE!

Faz sentido, para você?

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)
{
```

```c
    int numero1=1,
        numero2=2;

    char letra1='a',
         letra2='b';

    printf("numero1: \n");
    printf("Valor: %d\n", numero1);
    printf("Endereco na memoria: %d\n\n", &numero1);

    printf("numero2: \n");
    printf("Valor: %d\n", numero2);
    printf("Endereco na memoria: %d\n\n", &numero2);

    printf("letra1: \n");
    printf("Valor: %c\n", letra1);
    printf("Endereco na memoria: %d\n\n", &letra1);

    printf("letra2: \n");
    printf("Valor: %c\n", letra2);
    printf("Endereco na memoria: %d\n\n", &letra2);
    printf("Alterando os valores...\n\n");

    numero1=2112;
    numero2=666;

    letra1='A';
    letra2='B';

    printf("numero1: \n");
    printf("Valor: %d\n", numero1);
    printf("Endereco na memoria: %d\n\n", &numero1);
    printf("numero2: \n");
    printf("Valor: %d\n", numero2);
    printf("Endereco na memoria: %d\n\n", &numero2);
    printf("letra1: \n");
    printf("Valor: %c\n", letra1);
    printf("Endereco na memoria: %d\n\n", &letra1);
    printf("letra2: \n");
    printf("Valor: %c\n", letra2);
    printf("Endereco na memoria: %d\n\n", &letra2);

    return 0;
}
```

A função sizeof() e os blocos vizinhos de memória

No artigo introdutório desta seção sobre ponteiros nossa apostila de C, demos uma ideia sobre o que são ponteiros em C, além de falarmos sobre os endereços de memória.

Agora, nesse artigo, vamos entrar mais a fundo no estudo da memória e ver, de fato, onde as variáveis estão sendo declaradas e um importante relação entre ponteiros e variáveis que ocupam vários bytes.

- **Quando cada variável ocupa em memória: A função sizeof()**

Como dissemos, o C vê sua memória RAM como um vetor enorme de bytes, que vão desde o número 0 até o tamanho dela (geralmente alguns Gb).

Sempre que declaramos uma variável em C, estamos guardando, selecionando ou alocando um espaço de bytes desses, e dependendo do tipo de variável, o tanto de memória reservada varia.

Por exemplo, embora as variáveis do tipo *float* e *double* sejam usadas para representar números em sua forma decimal, as variáveis do tipo *double* têm, como o próprio nome sugere, o dobro de precisão. Ou seja, podemos colocar muito mais casas decimais em variáveis desse tipo. E para que isso aconteça, é óbvio que vamos precisar de um espaço maior em memória.

Podemos descobrir quantos bytes certa variável ocupa através da função *sizeof()*.
Essa função recebe uma variável como argumento, ou as palavras reservadas que representam as variáveis: char, int, float etc.

Como você poderá ver ao longo de nossa apostila de C, a função *sizeof()* é MUITO importante e será MUITO usada. Ela é tão importante que foi definido um novo 'tipo' de variável para sua saída.

Sempre que usamos a função *sizeof()*, ela retorna variáveis do tipo: ***size_t***
Lembre-se bem desse tipo. Vamos usar bastante na seção de *strings* e de *alocação dinâmica de memória.*

Vamos ver um exemplo que mostra o uso da sizeof.

- **Exemplo de código: Como saber quantos bytes cada variável ocupa em memória**

Faça um programa em C que mostra quantos bytes ocupam cada uma das variáveis: char, int, float e double.

Existem duas maneiras de fazer isso, a primeira é simplesmente colocando as palavras reservadas dentro da função *sizeof()*. A segunda maneira é declarando variáveis e colando ela dentro da função *sizeof()*, como faremos no próximo exemplo.

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)
{
	printf("Char: %d bytes\n", sizeof(char));
	printf("Int: %d bytes\n", sizeof(int));
	printf("Float: %d bytes\n", sizeof(float));
	printf("Double: %d bytes\n", sizeof(double));

	return 0;
}
```

- **Exemplo: Mostrar o endereço e número de bytes que cada variável ocupa**

Agora, além de mostrar quantos bytes cada variável ocupa, mostre o endereço dela.
Para isso, declare 4 variáveis: uma char, um int, um float e um double.

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)
{
	char caractere; int inteiro;
	float Float;
	double Double;

	printf("Caractere: %d bytes \t em %d\n", sizeof(caractere),
		&caractere); printf("Inteiro: %d bytes \t em %d\n",
		sizeof(inteiro), &inteiro);
```

```
	printf("Float: %d bytes \t em %d\n", sizeof(Float), &Float);
	printf("Double: %d bytes \t em %d\n", sizeof(Double), &Double);

	return 0;
}
```

- **Endereço de um bloco de variáveis**

Agora que você já viu o tamanho e localização que cada variável ocupa, vamos ensinar um fato importante, que será muito usado em nosso estudo de *strings* e na compreensão da passagem de parâmetro por referência, em funções.

Quando declaramos uma variável ou vetor de variáveis, os blocos de memória são vizinhos, são separados em fila, numa ordem.

Você viu no exemplo passado que, tirando o tipo *char*, as outras variáveis ocupam mais de uma posição em memória (lembrando que 1 byte é uma posição, ou bloco de memória).

O tipo *double,* por exemplo, ocupa 8 bytes. Então, esses 8 espaços de memória são alocados de forma contígua (um ao lado do outro).

Sempre que declaramos uma variável que ocupa mais de um bloco de memória, o seu endereço será o **endereço do primeiro bloco**. Ou seja, se um inteiro ocupa 4 bytes (4 blocos), e usamos o operador & para ver seu endereço, o endereço que vemos é o endereço do **primeiro** byte. E qual o endereço dos outros blocos? Ué, são vizinhos. Então, ocupam bytes, ou posições de memória vizinha.

Se o primeiro byte está em 2112, o segundo vai estar na posição 2113, o terceiro na posição 2114 e o quarto na posição 2115.

Vamos supor que queiramos declarar 1 milhão de inteiros em um vetor:
int numero_zao[1000000];

O endereço de 'numero_zao' é o endereço de 'numero_zao[0]', pois o endereço de uma variável ou vetor, é o endereço do primeiro bloco. E quantos bytes essa variável ocupa?
1 milhão * 4 bytes. = 4 000 000 bytes

Teste com : sizeof(numero_zao)

(Por ser um número muito grande de memória, há a possibilidade de sua máquina ficar lenta, de dar erro no programa, ou mesmo sua máquina travar. Lembre-se, a linguagem C é poderosa, você está manipulando e controlando cada pedacinho de sua máquina, por isso é fácil fazer alguma coisa que possa travar seu sistema. Não é à toa que C é a linguagem favorita dos hackers e a mais usada na criação de vírus).

O mesmo ocorre para um ponteiro. Sabemos que os ponteiros, ou apontadores, armazenam o endereço de **apenas um** bloco de memória. Quando um ponteiro aponta para uma variável que ocupa vários bytes, adivinha pra qual desses bytes o ponteiro realmente aponta? Ou seja, o endereço que o tipo ponteiro armazena, guarda o endereço de qual byte? **Do primeiro. Sempre do primeiro.**

E como ele sabe o endereço dos outros? Os outros estão em posições vizinhas!

Vamos mostrar isso na prática com um exemplo.
Para isso, temos que declarar duas variáveis de um mesmo tipo, e ver seus endereços de memória.

- **Exemplo de código: Mostrando as posições dos blocos de variáveis**

Crie duas variáveis de cada tipo: char, int, float e double - e mostre o endereço destas.

Ao declarar e ver os endereços, note que as variáveis do mesmo tipo, que foram declaradas juntas, estão em endereços de memória contínuos.

No caso das chars, elas estão ao lado da outra, pois só ocupam 1 byte cada. No caso dos inteiros e floats, o espaço de endereço de uma variável pra outra é de 4 unidades, pois cada variável desta ocupa 4 bytes.

Nas variáveis do tipo double, seus endereços estão distantes em 8 unidades, bytes, um do outro.

Rode o seguinte código e veja:

#include <stdio.h>

```
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)

{
    char caractere1, caractere2;
    int inteiro1, inteiro2;
    float Float1, Float2;
    double Double1, Double2;

    printf("Caracteres: %d e %d\n", &caractere1, &caractere2);
    printf("Inteiros: %d e %d\n", &inteiro1, &inteiro2);
    printf("Floats: %d e %d\n", &Float1, &Float2);
    printf("Doubles: %d e %d\n", &Double1, &Double2);
    return 0;

}
```

Como declarar, inicializar e usar ponteiros em C - A constante NULL

Agora que já vimos os conceitos teóricos sobre memória, blocos de memória, endereçamento e do uso da função *sizeof()*, vamos de fato usar os ponteiros.

Nesse tutorial de C de nossa apostila vamos ensinar como declarar ponteiros, fazê-los apontarem para alguma variável ou vetor, e manipulá-los.

- **Como declarar ponteiros em C**

Para declarar um ponteiro, ou apontador, em C basta colocarmos um asterisco - * - antes do nome desse ponteiro.

Sintaxe:
tipo *nome_do_ponteiro;

Por exemplo:
int *ponteiro_pra_inteiro;
float *ponteiro_pra_float;
char *ponteiro_pra_char;

Na verdade, esse asterisco pode ser encostado no tipo ou entre o tipo e o nome.

Aqui, se você estiver com os conceitos de ponteiro na cabeça, pode surgir uma pergunta.

“Se os ponteiros são tipos que armazenam endereço, e endereço são apenas números, por quê ter que declarar ponteiros com os tipos (int, float, char etc) ?”

A resposta é dada no artigo passado, em que falamos sobre o tamanho que as variáveis ocupam em memória.

Vimos que as variáveis ocupam posições vizinhas e contíguas (em seqüência) de memória (exceto, claro, o tipo char, que ocua só 1 byte, ou seja, só um bloco).

Vamos pegar o exemplo da variável inteira. Em minha máquina, ela ocupa 4 bytes.

Ou seja, 4 blocos de memória, cada bloco com um endereço.
Mas o ponteiro armazena **apenas um** endereço de memória, e não 4.
Então, o ponteiro **irá sempre armazenar o endereço do primeiro bloco, do primeiro byte**.

E os outros? Ué, se o C sabe quantos bytes cada variável ocupa, que elas são blocos vizinhos de memória e o ponteiro sabe o endereço do primeiro bloco, ele vai saber dos outros também!

É por isso que precisamos dizer o tipo de variável, antes de declarar o ponteiro.
Se for um ponteiro de inteiro, estamos dizendo: “Ponteiro, guarde esse endereço e os próximos 3, pois o inteiro tem 4 bloco”.
Se for um double: “Ponteiro, armazene o primeiro endereço, e saiba que os próximos 7 blocos são dessa mesma variável.”

- **Ponteiros e Vetores em C**

Já explicamos sobre a relação dos ponteiros com os diversos tipos de blocos de memória, de cada variável.
E a relação dos ponteiros com os vetores, que possuem diversas variáveis?

Pois bem, eles têm (ponteiros e vetores) possuem uma relação especial. Quando declaramos um vetor, estamos declarando um conjunto de variáveis também contíguas, e cada uma dessas variáveis ocupam vários bytes (ou só 1 byte, se for char). Então, um vetor é um conjunto maior ainda de bytes, de blocos de memória.

Como você sabe, quando apontamos um ponteiro para uma variável, esse ponteiro armazena o endereço do primeiro byte, do menor endereço, da variável.

A relação com vetores é análoga: **o nome do vetor é, na verdade, o endereço do primeiro elemento desse vetor**.

Ou seja, se declararmos um vetor de nome ***vetor***, não importando o número de elementos, se imprimirmos o nome ***vetor*** dentro de um printf, veremos o endereço da primeira variável daquele vetor.
Podemos ver um vetor como um ponteiro.

Isso explica o fato de que quando passamos um vetor para uma função, essa função altera de fato o valor do vetor. Isso ocorre pois não estamos passando uma cópia do vetor (como acontece com as variáveis).
Isso ocorre porque quando passamos o nome do vetor, estamos passando um ponteiro pra função.
Ou seja, estamos passando um endereço, onde a função vai atuar.
E endereço de memória é o mesmo, dentro ou fora de uma função.

Rode o seguinte exemplo para se certificar do que foi ensinado aqui.

- **Exemplo de código em C**

Crie um programa que mostre que o nome de um vetor é, na verdade, um ponteiro para a primeira posição que o vetor ocupa na memória. Ou seja, **um vetor sempre aponta para o elemento 0.**

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)

{
    int teste[10];

    printf("Imprimindo o vetor 'teste': %d\n", teste);
    printf("Imprimindo o endereço do primeiro elemento: %d\n", &teste[0]);
    return 0;

}
```

Ou seja, para declararmos um ponteiro ***ptr*** para um vetor ***vet[ ]***, fazemos:
ptr = vet;

Pois o nome do vetor é um ponteiro (que não muda) para o primeiro elemento.

Então poderíamos fazer assim também:
ptr = &vet[0];

- **Como inicializar um ponteiro em C – A constante NULL**

Já vimos como declarar um ponteiro, então é hora de fazer com que eles cumpram sua missão.
Vamos fazer os ponteiros apontarem.

Lembra que ensinamos como checar o endereço de uma variável ou vetor apenas usando o símbolo & antes da variável?
Agora vamos fazer isso novamente, mas é para pegar esse endereço e armazenar em um ponteiro.

Por exemplo, se quisermos armazenar o endereço do inteiro '*numero'* no ponteiro '*numeroPtr*', fazemos:

```
int numero = 5;
int *numeroPtr = &numero;
```

Pronto, agora nosso ponteiro está apontando para a variável *numero*, pois o ponteiro guardou o endereço do inteiro na sua posição de memória.
Muito cuidado! Ponteiros armazenam endereços, e não valores. Ou seja, se fizer:
int *numeroPtr = numero;

Estará comentendo um erro!

É sempre bom inicializarmos os ponteiros, pois senão eles podem vir com lixo e você se esquecer, posteriormente, de inicializar. Então, quando for usar, pensará que está usando o ponteiro de modo correto, mas estará usando o ponteiro com ele apontando para um lixo (endereço qualquer de memória).

Uma boa prática é apontar os ponteiros para a primeira posição de memória, que é conhecida como **NULL**. Sempre que terminar de usar um ponteiro, coloque ele pra apontar para a posição NULL. Para fazer isso, faça:
tipo *nome_do_ponteiro = NULL;

- **Exemplo de código: Como usar ponteiros**

Crie um programa em C que declara um inteiro e uma variável do tipo double. Em seguida, crie dois ponteiros apontando para essas variáveis e mostre o endereço de memória das variáveis, e mostre o endereço de memória que cada ponteiro armazenou.

Por fim, coloque esses ponteiros para a primeira posição (NULL), de memória.

Para saber o endereço de uma variável dentro do printf, colocamos o %d e depois '&nome_variavel'.
Para saber que endereço um ponteiro armazena no printf, também colocamos o %d entre as aspas, e fora colocamos apenas o nome do ponteiro.

Veja como ficou nosso código sobre como fazer esse programa em C:

```c
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)

{
    int inteiro;
    int *inteiro_ptr = &inteiro;
    double double1;
    double *double_ptr = &double1;

    printf("Endereco da variariavel 'inteiro': %d\n", &inteiro);
    printf("Endereco armazenado no ponteiro 'inteiro_ptr': %d\n\n",
inteiro_ptr);
    printf("Endereco da variariavel 'double1': %d\n", &double1);
    printf("Endereco armazenado no ponteiro 'double_ptr': %d\n\n", double_ptr);
    printf("Apos o uso dos ponteiros, vamos aponta-los para NULL\n\n");
    inteiro_ptr = NULL;
    double_ptr = NULL;
    printf("Endereco armazenado no ponteiro 'inteiro_ptr': %d\n", inteiro_ptr);
    printf("Endereco armazenado no ponteiro 'double_ptr': %d\n", double_ptr);
    return 0;
}
```

Variáveis apontadas - A Passagem por Referência em C

Já vimos como declarar e inicializar ponteiros em C, bem como estudamos a teoria por trás dos endereços de memória e blocos de memórias, vamos mostrar agora em nossa apostila a maior utilidade dos ponteiros, que é trabalhar com os valores das variáveis para qual eles apontam, e alterar seu valor.

- **Obtendo o valor apontado pelo ponteiro: ***

Para obtermos o valor da variável na qual o ponteiro aponta, devemos colocar um asterisco - * - antes do ponteiro, assim, o ponteiro irá mostrar o valor da variável (a variável que ele aponta), e não mais seu endereço.

Por exemplo, vamos supor que tenhamos declarado a variável inteira '*numero*':
int numero = 1;

Agora vamos criar um ponteiro '*ptr_int*' e fazê-lo apontar para '*numero*':
int *ptr_int = №

Pronto, agora *ptr_int* aponta para *numero.*
Para saber o valor de *numero* através do ponteiro, usamos: **ptr_int*

Veja bem:
ptr_int -> armazena o endereço da variável *numero*
*ptr_int -> se refere ao VALOR da variável, ou seja, ao valor da variável *numero*.

- **Exemplo de código: Mostrando o valor das variáveis apontadas por ponteiros**

Crie um programa em C que peça ao usuário três números inteiros e armazene em três variáveis inteiras através do uso de um ponteiro.
Após o usuário inserir cada número, mostre o número exibido, porém mostre através do ponteiro.

Desde o início de nosso curso de C, quando ensinamos como obter dados a partir do usuário, através da função *scanf()*, nós vínhamos usando o

operador & dentro dessa função para que o dado inserido pelo usuário fosse armazenado em determinada posição de memória.

Aqui, vamos criar três inteiros e um ponteiro de inteiro, que irá apontar para cada uma dessas variáveis. Usaremos o ponteiro dentro da scanf(uma vez que ele guarda o endereço da variável pra onde ele aponta) e o asterisco, para mostrar o valor da variável que é apontada por ele.

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)

{
    int int1, int2, int3;
    int *ptr_int = &int1;

    printf("Inteiro 1: ");
    scanf("%d", ptr_int);
    printf("Numero inserido: %d\n", *ptr_int);

    ptr_int = &int2;
    printf("Inteiro 2: ");
    scanf("%d", ptr_int);
    printf("Numero inserido: %d\n", *ptr_int);

    ptr_int = &int3;
    printf("Inteiro 3: ");
    scanf("%d", ptr_int);
    printf("Numero inserido: %d\n", *ptr_int);
    return 0;

}
```

- **O que é e como fazer a Passagem por referência em C**

Para passarmos uma variável para uma função e fazer com que ela seja alterada, precisamos passar a referência dessa variável, em vez de seu valor.

Até o momento, vínhamos passando somente o valor das variáveis. Conforme explicamos, quando passamos um valor, a função copia esse valor e trabalhar somente em cima da cópia dessa variável, e não na variável em si. Por isso nossas variáveis nunca eram alteradas quando passadas para funções.

Porém, é muito importante e útil que algumas funções alterem valores de variáveis.

Para fazer isso, usaremos um artifício chamado Passagem por Referência. Por referência entenda endereço, um local. No caso, em vez de passar o valor da variável, **na passagem por referência vamos passar o endereço da variável** para a função.

Para fazer isso, basta colocar o operador & antes do argumento que vamos enviar, e colocar um asterisco * no parâmetro da função, no seu cabeçalho de declaração, para dizer a função que ela deve esperar um endereço de memória, e não um valor.

Sim, parece confuso e difícil entender assim de cara, mas vamos mostrar dois exemplos em que usaremos a passagem por referência.

- **Exemplo de código: Passagem por referência em C**

Crie um programa que recebe um inteiro e dobra seu valor.

Para que uma função altere o valor de uma variável, é necessário que essa função atue no endereço de memória, e não na copilar do valor. Para isso, temos que passar o endereço da variável para a função (usando &), e a função tem que ser declarada de modo a esperar um ponteiro (usando *).

Dentro da função, devemos trabalhar com o valor na qual aquele ponteiro aponta.
Ou seja, se o ponteiro tiver nome ‘*ptr*’, devemos trabalhar com o valor ‘**ptr*’.

Veja como ficou nosso código C:

```c
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

void dobra(int *num)
{
    (*num) = (*num) * 2;
}

int main(void)
```

```
{
    int num;

    printf("Insira um numero: ");
    scanf("%d", &num);

    dobra(&num);
    printf("O dobro dele eh: %d\n", num);
    return 0;

}
```

- **Exemplo de código: Trocar o valor de dois números em C**

Crie um programa em C que peça ao usuário o valor de uma variável ***x*** e depois de uma ***y***.
Crie uma função que inverta esses valores através do uso de ponteiros.

Mais uma vez, declaramos uma função que irá receber dois ENDEREÇOS de memória, ou seja, irá receber dois ponteiros. Esses ponteiros têm os locais onde as variáveis ***x*** e ***y*** foram armazenadas.

Para alterar esses valores, vamos trabalhar com ***(*x)*** e ***(*y)***.

Para fazer duas variáveis trocarem de valores entre si, iremos precisar de uma variável temporária.

Primeiro guardamos o valor de ***x*** na variável temporária (vamos precisar depois).

Em seguida, fazemos ***x*** receber o valor de ***y***.
Agora é ***y*** que tem que receber o valor de ***x***. Mas x mudou de valor, lembra? Ou seja, precisamos pegar o antigo valor de ***x***. Foi por isso que guardamos esse antigo valor de ***x*** na variável temporária.

Logo, agora é só fazer com que ***y*** pegue o valor da variável temporária, e teremos os valores invertidos.

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

void troca(int *x, int *y)
{
```

```c
    int tmp;

    tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void)

{
    int x, y;

    printf("Insira o numero x: ");
    scanf("%d", &x);

    printf("Insira o numero y: ");
    scanf("%d", &y);

    troca(&x, &y);

    printf("Agora x=%d e y=%d\n", x, y);
    return 0;

}
```

Operações Matemáticas com Ponteiros em C

Agora que ensinamos como fazer os ponteiros apontarem para variáveis, bem como ver e alterar o valor de variáveis através dos ponteiros, neste tutorial de nossa **apostila de C**, vamos dar mais exemplos reais de uso dos ponteiros, com operações de incremento, decremento, comparação de igualdade e negação.

- **Incremento, Decremento e Comparando ponteiros em C**

Uma das maiores utilidades dos ponteiros é fazer com que eles apontem para quantas variáveis desejarmos.
Por exemplo, podemos fazer com que um único ponteiro percorra todas as variáveis de um vetor.

E através do operador asterisco, além de percorrer as variáveis, também podemos fazer operações com elas.

A vantagem de usarmos ponteiros para percorrer e alterar variáveis é a eficiência, pois o C trabalha bem mais rápido se usarmos endereços de memória (ponteiros), ao invés de ficarmos declarando e alterando valores de variáveis.

Além disso, o uso de ponteiros para o estudo de Arquivos, Alocação Dinâmica de Memória e de Strings, na próxima seção, é essencial em nosso curso de C.

Assim como outras variáveis, podemos, por exemplo, incrementar e decrementar ponteiros.
Como os ponteiros são variáveis para armazenar endereços de memória, se fizermos operações de incremento e decremento em ponteiros, é o endereço que irá mudar.

Porém, quando incrementamos ou decrementamos um ponteiro, seu endereço não percorre os endereços em uma unidade, ele percorre em *sizeof(tipo_variável).*

Por exemplo, sabemos que o tipo *double* ocupa 8 bytes em memória.
Se tivermos um ponteiro para um vetor tipo de variável, ele guarda a primeira posição do vetor.

Vamos supor que esse endereço seja: 2112
Quando incrementamos o ponteiro: *ptr++;*
O valor que ele aponta não será o pro endereço 2113, e sim pro endereço 2120, pois a próxima variável do vetor vai estar na posição 2120, já que cada variável ocupa 8 bytes e a primeira estava na posição 2112.

Ou seja, quando incrementamos ou decrementamos um ponteiro, que aponta para um tipo, ele vai percorrer o número de bytes do tamanho de seu tipo.
Se for *char*, o ponteiro muda sua posição de 1 byte para frente ou para trás no endereço.
Se for *int*, o ponteiro muda sua posição em 4 bytes, para frente ou para trás no endereço.

O mesmo vale para outras operações matemáticas, como soma e subtração:
ptr + 1 -> endereço de onde *ptr* apontava + *sizeof(tipo)*.
ptr + 2 -> endereço de onde *ptr* apontava, deslocado de duas variáveis para frente.
ptr – 3 -> endereço de onde *ptr* apontava, deslocado de duvas variáveis para trás

- **Exemplo de código: Calcular os termos de uma P.A usando Ponteiros em C**

Peça ao usuário dois inteiros, onde o primeiro será o primeiro termo de uma P.A (Progressão Aritmética) e o segundo a razão. Em segundo, através de ponteiros, preencha um vetor com os 10 primeiros elementos dessa P.A.

Após declararmos um vetor de 10 elementos, fazemos com que o ponteiro aponte para esse vetor. Como o nome vetor nada mais é que do que o endereço do primeiro elemento.

Assim, inicialmente, nosso ponteiro aponta para o primeiro elemento do vetor, que será o primeiro elemento da P.A.

Para calcular os elementos da P.A, basta fazermos com que o próximo termo seja o atual mais a razão:
*(ptr+1) = *ptr + razao

Entramos no laço WHILE que irá percorrer todos os elementos do vetor, exceto o último (pois o último elemento é calculado pelo penúltimo, então não precisa ir até o último).

Ou seja, só irá parar depois de termos atingidos o penúltimo elemento do vetor, que é *pa[8]*.

Como vamos usar somente ponteiros para percorrer o vetor, nosso WHILE deve continuar enquanto o endereço de nosso ponteiro for diferente último endereço do vetor (que não precisará ser acessado, pois cada iteração do looping calcula o próximo termo da PA).

Em seguida, usamos o laço FOR para você ver como podemos manipular ponteiros dentre dele também.

Inicializamos, novamente, o ponteiro a partir do primeiro elemento: ptr = pa;
Depois, imprimimos todos os elementos, enquanto o endereço que o ponteiro aponta for menor ou igual ao último endereço do vetor.

Veja como ficou nosso código:

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre a linguagem de programação C !
int main(void)

{
    int pa[10], razao;
    int *ptr;


    printf("Insira o termo inicial da PA: ");
    scanf("%d", &pa[0]);
    ptr = pa;

    printf("Insira razao da PA: ");
    scanf("%d", &razao);
    while( ptr != &pa[9])
    {
        *(ptr+1) =  *ptr + razao;
        ptr++;
    }
    for(ptr = pa ; ptr <= &pa[9] ; ptr++)
        printf("-> %d ", *ptr);
    return 0;
}
```

Na próxima seção, vamos usar todos esses conceitos sobre ponteiros com bastante frequência.

Strings

Nessa seção de nosso curso, vamos apresentar os principais conceitos, funções e bibliotecas que nos permitem trabalhar com caracteres em linguagem C.

Strings servem para trabalharmos com textos. Pode parecer simples, mas não é, além de ser essencial dominar os conceitos de strings.

Constantemente, em aplicativos C, precisamos mostrar textos, comparar caracteres, extrair informações, armazenar nomes de arquivos, URL's, validar datas, números de CPF, RG e milhões de outras coisas necessitam do completo domínio de strings em C.

Diferente das outras linguagens, as strings em C exigem um cuidado especial, pois trabalhamos caractere à caractere, posição por posição do vetor. Temos que ter total controle de todas as variáveis do tipo *char*.

Como costumamos dizer e mostrar, em C você vai sempre mais à fundo.

Introdução as strings

Após nosso estudo sobre vetores e das noções sobre ponteiros, vamos usar esses conhecimentos obtidos em nosso curso online de C.

Tente achar um programa que não tenha nada escrito. Que não seja necessário ler ou informar alguma letra ou palavra.
Até nas calculadoras temos que inserir caracteres (+, -, *, % etc).

Essa seção é totalmente dedicada a escrita, suas bibliotecas, funções, funcionamento e detalhes. Sim, vamos aprender a escrever em C, nesta seção de nossa **apostila**.

- **O que são *Strings* em linguagem C**

Não foi à toa que ensinamos vetores antes de ensinarmos strings.
String é um vetor de caracteres com um delimitador que indica o final da string: \0

Por exemplo, para escrever “ C Progressivo”, seria necessário declararmos 13 caracteres e preenchê-los um por um, o que, obviamente é inviável, pois daria muito trabalho.
Imagine escrever textos longos com variáveis do tipo *char* puras.

Para contornar isso, o C trata vetores de caracteres de uma forma muito especial, com certas ‘regalias’, detalhes e opções, em relação aos outros tipos de vetores.

Na verdade, já vínhamos usando strings em nosso curso de C, dentro da função *printf*.
Podemos diferenciar as strings dos caracteres, porquê as strings aparecem dentro de aspas duplas, e os caracteres dentro de aspas simples.

“Curso C Progressivo” -> String
“C” -> String
‘C’ -> Caractere

- **O caractere especial delimitador de strings: \0**

Antes, porém, de ensinarmos a declarar, inicializar e usar strings em C, temos que falar sobre esse caractere especial: \0

Esse caractere é o 0 do código ASCII (não é o caractere 0, é o elemento de número 0 do código ASCII), e é o delimitador de final de string.
Ou seja, ele representa o fim de uma string.

Se não tiver o delimitador ao fim de um vetor de caracteres, não é string, é apenas um vetor de caracteres (sim, são coisas diferentes).

- **Como declarar e inicializar Strings em C**

Para declarar string em C, não há segredo, pois é a mesma coisa de declarar um vetor de variáveis do tipo *char*.

A sintaxe é sempre a seguinte:
```
char nome_da_string[tamanho];
```

Aqui, porém, vale uma ressalva muito importante que geralmente os programadores C iniciantes esquecem: o caractere delimitador - \0 – também fará parte da string.
Ou seja, ele conta no número de caracteres, no tamanho da String.

Por exemplo: “C Progressivo”
Quantos caracteres existem? 13 não é? Mas o número total de caracteres é 14, pois o caractere \0 também vai ‘comer’ uma variável do tipo char.

Então, para declarar uma string que vai armazenar “C Progressivo”, temos que fazer:
```
char curso_de_C[14];
```

Logo, sempre que formos declarar uma string, temos que levar em consideração no seu tamanho o caractere delimitador. No exemplo que demos, temos que declarar uma string que armazene 14 caracteres ou mais.

- **Como inicializar strings em C**

Lembrando que as strings estão entre aspas, podemos ter declarado e inicializado essa string assim:
char curso[14] = “C Progressivo”;

E o caractere delimitador? O C coloca, automaticamente, ao fim da string.
Podemos fazer assim também:
char curso[] = “C Progressivo”;

Que lembra o que fazíamos nos vetores. Desse jeito, o C vai alocar o número exato de caracteres necessários, que no caso é 14 (13 para nossa frase mais 1 para o \0).

Outra possibilidade de carga automática, onde o C aloca o número certo de caracteres, é através de ponteiros. Lembra que dissemos que o nome do vetor é, na verdade, um endereço de memória? Endereço do primeiro elemento do vetor?
Podemos então inicializar uma string através de um ponteiro para o tipo char:
char *curso = “C Progressivo”;

Quando formos estudar e criar as funções da biblioteca *string.h*, que nos fornece diversas opções para trabalhar com strings, vamos usar bastante os conceitos de ponteiros.

Por fim, podemos inicializar as strings da maneira mais ‘trabalhosa’, que é caractere por caractere onde,
mais uma vez, o C colocar o delimitador de string \0 na última posição:
char curso[14] = {‘C’, ‘ ‘, ‘P’, ‘r’, ‘o’, ‘g’, ‘r’, ‘e’, ‘s’, ‘s’, ‘i’, ‘v’, ’o’};

E se declararmos com um tamanho maior que o texto? Por exemplo:
char curso[2112] = “C Progressivo”;

Da posição 0 até a 12, existirão os caracteres da frase.
Na posição 13, o C coloca o caractere delimitador, sinalizando o final da string.
E nas outras posições? Lixo.

- **Exemplo de código: Declarando, Inicializando e Exibindo strings em C**

Declare uma string e inicialize. A seguir, exiba essa string na tela.

Vamos declarar uma string de 20 caracteres.
Podemos preencher a string com que caractere quisermos, mas a última posição da string é sempre o delimitador \0.
O C coloca esse delimitador sempre após o último caractere inserido.

Como nosso vetor tem 20 posições, podemos preencher, no máximo, até a posição 18.

Como usamos somente a frase “C Progressivo”, de 13 caracteres, o caractere \0 será o 14º caractere.
Vamos usar o primeiro laço for para imprimir todos os caracteres da string ‘*curso*’.

Note que o C imprime todos os caracteres antes do \0, e depois dele, nenhum.

O segundo laço for faz com que nossa variável auxiliar inicie na posição 0 da string e vá até a posição em que encontra o \0. Quando encontra, o laço para e mostramos em que posição encontrou o delimitador.

```c
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Artigos, apostilas, tutoriais e vídeo-aulas sobre
// a linguagem de programação C !

int main(void)
{
    char curso[20] = "C Progressivo";
    int count;

    for(count=0 ; count < 20 ; count++)
        printf("%c",curso[count]);

    for(count=0 ; curso[count] != '\0' ; count++)
            ;
    printf("\nO '\\0' esta na posicao %d da string\n", count);
    return 0;

}
```

Na verdade, existem maneiras bem mais simples de se exibir strings, que você aprenderá no próximo tutorial de nosso curso de C.
Decidimos ensinar dessa maneira para você saber o que ocorre por debaixo dos panos, ver o caractere delimitador atuando de verdade.

Lendo e Escrevendo Strings em C

Que 99,99% dos aplicativos em C, ou de qualquer outra linguagem, usam strings e caracteres para mostrar textos, nós já convencemos você.

Mas só mostrar não adianta muita coisa, geralmente é preciso receber strings do usuário.

Afinal, quem nunca forneceu o nome pra ficar no ranking daquele jogo, data e local de nascimento, nome dos pais e outros tipos de texto?

Nesse tutorial sobre strings em C, vamos ensinar como receber textos do usuário.

- **Strings em C: %s**

Antes de iniciar nosso estudo das funções que recebem e lêem strings, vamos apresentar o %s.
Você deve se lembrar que usamos %d para representar números inteiros, %f para números decimais e %c para caracteres.

Para strings, usamos o %s.

- **Como ler strings com a função *scanf()***

Assim como os números e caracteres, também podemos usar a nossa tão conhecida função scanf para receber uma string do usuário.

Há um porém com a leitura de strings e um problema com a função scanf para strings.

O porém é o seguinte: lembra que repetimos, exaustivamente, que o nome de um vetor nada mais é que o endereço de memória do primeiro elemento do vetor?

Lembra que dissemos também no tutorial de C passado, sobre a definição de strings, que strings são um vetor de caracteres com um caractere delimitador \0 no final?

E por fim, lembra que, para armazenar dado em uma variável, na função *scanf()* nós sempre passamos o endereço da variável (&variável)?

Se você conseguiu ver onde chegamos, notará que para receber uma string do usuário através da função *scanf()*, não é necessário colocar o operador &, pois o nome da string em si já é um endereço de memória.

A sintaxe para receber uma string por meio da *scanf()* é:
scanf(“%s”, nome_da_string);

- **Exemplo: Programa que pede o nome e sobrenome em C**

Crie um aplicativo em C que peça ao usuário seu nome, armazene em uma String, peça o sobrenome, armazene em outra string e exiba o nome do usuário de maneira formal (Sobrenome, Nome).

Criamos duas strings, optamos por colocar 20 caracteres em cada, como temos que ter o caractere delimitador, então declaramos o vetor de caracteres com 21 elementos cada, um pro nome e outro pro sobrenome.

Depois pedimos o nome ao usuário, armazenando na variável ‘nome’, e fazemos o mesmo na variável ‘sobrenome’.
Em seguida, para exibir a string completa, basta usarmos o símbolo %s , da mesma maneira que vínhamos fazendo com %d, %f e %c na *printf()*

Veja como ficou o código de nosso programa em C:

```c
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Apostila online, tutorial completo sobre
// a linguagem de programação C !

int main()
{
    char nome[21], sobrenome[21];

    printf("Primeiro nome: ");
    scanf("%s", nome);

    printf("Ultimo sobrenome: ");
    scanf("%s", sobrenome);

    printf("Ola senhor %s, %s. Bem-vindo ao curso C Progressivo.\n", sobrenome,
nome);
    return 0;
}
```

Vamos agora apontar o problema do uso da função *scanf()* receber dados do usuário.

Você deve ter digitado apenas um nome e um sobrenome.

Pois bem, digite um nome composto no ‘nome’ ou no ‘sobrenome’. Ou seja, um nome com espaço em branco.

A *scanf()* vai simplesmente cortar seu nome composto. Essa função pega tudo **até** encontrar um espaço em branco, caractere *new line* \n, tab ou ENTER.
Para corrigir isso, o C tem uma função especial e bem mais simples para receber strings do usuário, a função *gets()*.

- **Como receber strings do usuário com a função *gets()***

gets vem de *get string*. Como sempre, algo bem óbvio e de fácil memorização.

Para usar a função *gets()*, é bem simples, basta passar uma string como argumento.
A sintaxe é:
gets(nome_da_string);

Mais uma vez, nunca use & quando for armazenar uma string. Isso não é necessário, pois string é um vetor, e o nome do vetor já é um endereço, e um endereço é o operador & seguido do nome da variável.

Como exibir strings com a função *puts()* e *printf()*

Já que mostramos como obter, facilmente, uma string do usuário a partir da função *gets()*, nada mais justo do que existir uma função que exibe uma string.

Essa função existe e é a *puts()* (de *put string*), e sua sintaxe é idêntica a da *gets():*
puts(nome_da_string_a_ser_exibida);

A nossa velha e conhecida ***printf*** também serve para exibir strings.

Como dissemos, as strings são caracterizadas por %s.
Então, para exibir uma string "str" com o printf, fazemos:
printf("Minha string: %s", str);

- **Exemplo de código: Programa em C que pede os dados cadastrais completos**

Crie um aplicativo em C que peça o nome do usuário, sua idade e data de nascimento.

Esse aplicativo é bem simples, e certamente você fará sem maiores problemas.
Porém, existe uma pequena casca de banana nele, que colocamos de propósito.

Vimos que a função *scanf()* pega tudo até aparecer o primeiro espaço em branco, e pára antes dele.
Já a *gets()* não, ela pega tudo até aparecer uma *new line* \n, inclusive nada.
Ou seja, se você der um ENTER, a *gets()* vai armazenar esse enter na string.

Note que após digitar o inteiro correspondente a idade, você dá um enter.
Esse número vai pra variável 'idade', mas e o ENTER, pra onde vai?
Vai pro buffer.

O problema é que a função *gets()* vai pegar o que está armazenado nesse buffer e vai armazenar o que estiver lá na string de data de nascimento!
E como evitar isso? Ora, é só apagar esse ENTER que está no buffer, usando o *fflush(stdin)* caso use Windows, ou *__fpurge(stdin)* caso seja abençoado e use Linux.

Então nosso código em C fica:

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Apostila online, tutorial completo sobre
// a linguagem de programação C !

int main(void)

{
    char nome[31], sobrenome[31], nascimento[11];
    int idade;
```

```
    printf("Nome: ");
    gets(nome);

    printf("Sobrenome: ");
    gets(sobrenome);

    printf("Idade: ");
    scanf("%d", &idade);

    fflush(stdin);

    printf("Data de nascimento: ");
    gets(nascimento);

    printf("\nNome completo: %s %s\n", nome, sobrenome);
    printf("Idade: %d\n", idade);
    printf("Data de nascimento: "); puts(nascimento);
    return 0;

}
```

- **Problemas com a gets() - A função fgets()**

Como C é uma linguagem de programação de baixo nível, algumas coisas são um pouco 'chatinhas' de serem usadas, pois se exigirá um rigor muito grande, que é pouco visto em outras linguagens de programação.

Um exemplo disso, que já mostramos neste mesmo artigo de nossa apostila, foi o da ***scanf()***, que só lê até o primeiro espaço, pois isso é a característica padrão da função.

É característica padrão, e não obrigatória. Ou seja, embora poucos saibam, é possível alterar o funcionamento da ***scanf()***. Por exemplo, se quisermos ler strings que tenham espaço, nós temos que dizer isso dentro da função.

Vamos dizer para a scanf() parar de pegar nossa string somente quando encontrar um caractere de NEW LINE (um enter). Para isso, usamos o operador: [^\n]

Logo, nosso código da ***scanf()*** para ler strings com espaços e armazenar na variável "str" é:
scanf ("%[^\n]", str);

(lembre-se de limpar o buffer, usando __fpurge(stdin)).

Podemos ainda limitar o tamanho de nossa string, basta colocar um numero inteiro ao lado do %, representando o número de caracteres máximo, o que é

uma excelente prática, pois essa função pode ocasionar problemas na memória, caso você estoure os limites da string.
Por exemplo:
scanf ("%256[^\n]", str);

A função ***gets()*** peca nesse quesito, de tamanho da string, pois podemos digitar mais caracteres do que a string alocou de memória, e "quebraríamos" o programa por conta de um *overflow*.
Algumas pessoas relatam problemas de usar ela em um ambiente Linux.

Uma solução para isso é usar a função ***fget()***, que é mais segura.
Ela recebe três dados: a string que vai armazenar o que vai ser digitado (no nosso caso é a variável "str"), o tamanho da string e de onde vai ler (ela pode ler de um arquivo de texto, por exemplo).
Para ler do teclado, usamos ***stdin***.

Veja como ficaria
fgets(str, 256, stdin);

- **Que função usar para ler strings, então?**

Se você está iniciando seus estudos e fazendo programas simples, apenas para aprender a linguagem C, pode usar a gets() ou scanf(), sem problemas.

Como dissemos, elas podem ocasionar problemas na memória, onde é fácil extrapolarmos o limite da memória, encerrando o programa por isso.
Portanto, se quiser usar uma função de maneira segura, em programas profissionais, evite a gets(). Use fgets() ou scanf() sempre com litadores de new line e tamanho da string.

Para saber mais sobre assunto, leia:
http://linux.die.net/man/3/gets
http://stackoverflow.com/questions/3302255/c-scanf-vs-gets-vs-fgets
http://stackoverflow.com/questions/1252132/difference-between-scanf-and-fgets

Como criar uma biblioteca em C

Vamos dar uma pausa em nossa apostila de C, especificamente sobre na seção de strings, para ensinar uma importante lição de organização.

Vamos aprender a criar e usar uma biblioteca, um header, que são aqueles arquivos de extensão .h que usualmente colocamos após o *#include*.

- **Por que criar uma biblioteca / header (.h) em C?**

Sempre que você faz:
#include <stdio.h>

É como se tivesse colocando os códigos que se encontram dentro de um arquivo chamado *'stdio.h'* no seu programa. Mas você não vê esses códigos, é o C/compilador que fazem isso tudo pra você.

Agora imagine se esses códigos aparecessem para você...sabe o que você veria?
Centenas de linhas de código para mostrar um simples *'Hello World'* ou outros programas simples, que vínhamos criando até agora.

A utilidade dessas bibliotecas e *headers* é a de criar um arquivo que contém diversas funções específicas, separadas e organizadas por assuntos.
Por exemplo, existe uma biblioteca chamada *string.h* (que iremos estudar em breve), que é armazena diversas linhas de códigos com funções prontas para manipularmos strings em C.

- **Como criar uma biblioteca / header (.h) em C**

Vá no Code::Blocks -> Empty File

Salve esse arquivo como: **mystring.h**
Salve na mesma pasta em que guarda seu código-fonte de C.
Escreva uma simples função nela, como uma que mostre uma mensagem na tela, com um simples printf e salve o arquivo.

Agora volte no seu projeto, que tem extensão .c
Após o *#include <stdio.h>*
Coloque:

#include "mystring.h"

Pronto. É como se todo o código contigo no arquivo *mystring.h* estivesse nesse seu arquivo de extensão .c
Faça um teste, invoque a função que criou no *header* a partir do seu arquivo .c

Note que só podemos usar o nome do *header* entre aspas se ele estiver na mesma pasta que seu código-fonte .c
Se estiver em outra como em: C:\C_Progressivo\, faça:
#include "C:\C_Progressivo\mystring.h"

Vamos criar uma série de funções de manipulação de caracteres e strings nos próximos tutoriais de nosso curso, e armazenaremos todos nossos códigos em bibliotecas!

A biblioteca string.h e suas funções

Trabalhar com *strings*, em C, não é nada óbvio, precisamos treinar bastante até ter total controle dos caracteres de nosso texto.

Essa dificuldade se torna ainda maior se você já tiver estudado outra linguagem de programação, como Java, Perl ou Python, onde a manipulação de texto é extremamente óbvia e simples.
Mas essa simplicidade toda não vem de graça, requer custos: processamento e memória de um computador.

Vamos agora, em nossa **apostila de C**, apresentar as principais funções para manipulação de *strings*, como usá-las e melhor: como criá-las! Sim, vamos criar funções de manipulação de *strings* para você saber como tudo funciona em C e como exercícios.

- **A biblioteca string.h e suas funções: quais existem, para quê servem e como usá-las**

Vamos agora mostrar as funções da biblioteca string.h e o que cada uma faz. Embora essas funções já existam e sejam testadas milhões de vezes a cada instante, vamos refazê-las.

“Mas para quê inventar algo que já existe?”
Simples: para aprender como funciona.
Você só aprende algo de verdade, quando faz a coisa. Só ver ou decorar como faz, não serve de absolutamente nada.

Você pode muito bem ler aqui as funções que existem, para quê servem e como usá-las.
Mas se quer realmente aprender a trabalhar com vetores, ponteiros e strings, sugerimos que tente fazer funções e colocar em sua biblioteca **mystring.h**
É isso o que vamos fazer nos próximos tutoriais.

Segue a lista das funções da biblioteca **string.h** cuja fonte é a Wikipedia: http://en.wikipedia.org/wiki/C_string_handling

Funções de exame de strings:

- **strlen: size_t strlen(const char *str);**

Essa função recebe um ponteiro que indica uma string e retorna quantos caracteres essa string possui.

- **strcmp: int strcmp(const char *lhs, const char *rhs);**

Essa função compara duas strings segundo sua ordem alfabética e retorna um inteiro.
Se esse inteiro for negativo, é porque a primeira string é menor que a segunda.
Se retornar um inteiro positivo, é porque a segunda string é maior que a segunda.
Se retornar 0, é porque as strings são idênticas.

- **strncmp: int strncmp(const char *lhs, const char *rhs, size_t count);**

Faz a mesma comparação da strcmp, mas ao invés de comparar toda a string, compara somente os 'count' primeiros caracteres.

- **strchr: char *strchr(const char *str, int ch);**

Retorna um ponteiro para a localização em que o caractere 'ch' aparece na string pela primeira vez na string apontada por *str, ou NULL se não encontrar.

- **strrchr: char *strrchr(const char *str, int ch);**

Faz a mesma coisa da função anterior, mas ao invés de localizar a primeira ocorrência de 'ch', localiza e retorna a última ocorrência.

- **strspn: size_t strspn(const char *dest, const char *src);**

Retorna o tamanho máximo do primeiro segmento na string 'dest' que consiste de elementos da string 'src'.

- **strcspn: size_t strcspn(const char *dest, const char *src);**

É o contrário da anterior, ou seja, retorna o tamanho máximo do segmento inicial na string 'dest' que consiste somente de elementos que NÃO ESTÃO na string 'src'.

- **strpbrk: char* strpbrk(const char* dest, const char* str);**

Retorna o primeiro caractere que está em ambas strings recebidas, ou NULL se não existe caractere comum.

- **strstr: char *strstr(const char* str, const char* substr);**

Retorna um ponteiro que indica a primeira ocorrencia da string 'substr' na string 'str'.

Funções de manipulação de strings

- **strcpy: char *strcpy(char *dest, const char *src);**

Copia a string 'src' para a string 'dest', e retorna um ponteiro para 'dest'.

- **strncpy: char *strncpy(char *dest, const char *src, size_t count);**

Faz a mesma coisa da strcpy, mas em vez de copiar todos os elementos, copia somente os 'count' primeiros elementos da 'src' para a 'dest'.

- **strcat: char *strcat(char *dest, const char *src);**

Copia (concatena) a string 'src' ao final da string 'dest'.

- **strncat: char *strncat(char *dest, const char *src, size_t count);**

Copia (concatena) 'count' elementos da string 'src' no final da string 'dest'.

- **memset: void* memset(void* dest, int ch, size_t count);**

Coloca o caractere 'ch' nas 'count' primeiras posições da string 'dest'.

- **memcpy: void* memcpy(void* dest, const void* src, size_t count);**

Copia os 'count' primeiros caracteres da string 'src' e coloca nas primeiras 'count' da string 'dest'.

- **memcmp: int memcmp(const void* lhs, const void* rhs, size_t count);**

Compara os 'count' primeiros caracteres das strings.
Retorna negativo se os 'count' primeiros caracteres da primeira string forem menor que a segunda.

Retorna positivo se os 'count' primeiros caracteres da primeira string forem maior que a segunda.

Retorna 0 se os 'count' primeiros caracteres das duas strings são iguais.

- **memchr: void* memchr(const void* ptr, int ch, size_t count);**

Acha a primeira ocorrência do caractere 'ch' nos primeiras 'count' elementos da string 'ptr' e devolve o endereço da ocorrência ou NULL.

Como implementar as funções da biblioteca string.h

Nós falamos, aqui em nossa **apostila de C**, no tutorial sobre as funções da biblioteca *string.h*, das principais funções e como usá-las.

Porém, é um excelente exercício tentar implementá-las!
Isso mesmo, tente criar sua própria biblioteca em C, que trata strings.
Fazendo isso, garantimos que você terá total domínio sobre as tão 'temidas' strings, em C.

- **Exercícios resolvidos sobre strings em C**

Para ver os enunciados das questões, veja como funcionam as funções:
A biblioteca string.h e suas funções

Muitos dizem que inventar algo que já existe é perda de tempo.
Mas nós, do curso C Progressivo, dizemos o contrário: você só realmente aprende uma coisa quando tenta fazer.

É muito fácil ler um livro ou assistir uma vídeo-aula, você pode até entender tudo que está lendo ou o que estão explicando.
Mas só vai dominar e criar programas quando você tentar, quando você raciocinar.

Só se aprende a programar programando. Ao final das seguintes implementações, você verá como foi o útil o processo de tentativa de criação de tais funções.

Óbvio que, profissionalmente, utilize a biblioteca *string.h* padrão, e não a sua. Vamos recriar essa biblioteca apenas para fins didáticos.

- **int strlen(char *str) - Calcule quantos caracteres uma string possui**

Lembrando que string é tudo aqui que vem antes do caractere delimitador \0, podemos calcular quantos caracteres uma string tem fazendo um laço percorrer todos seus elementos, desde o de índice 0 até o índice que do último caractere que não é o \0.

Para descobrir onde está o primeiro \0, basta fazermos a comparação de cada caractere com o \0.
Se o caractere comparado não for \0, incrementamos uma variável de contagem.
Ao final do looping, essa variável terá o número de caracteres da strings.

É bem simples, veja como fica nosso código C:

```
int strlen(char *str)
{
    int total=0;

    while( str[total] != '\0')
        total++;

    return total;
}
```

- **char *strcpy(char *dest, char *orig) - Copie a string *orig* para a string *dest***

Nesse exercícios, temos que copiar todos os caracteres da string de origem para a string de destino.

Ou seja, vamos copiar do elemento de índice 0 da string de origem, até o índice que contém o delimitador \0.

Para isso, podemos usar a função *strlen()*, que retorna o número de caracteres da string.
Note que, os caracteres visíveis na string de origem, são de 0 até *strlen(orig) -1*
E o caractere de índice *strlen(orig)* é o caractere delimitador \0.

Por isso, vamos copiar o caractere da posição 0 até o da posição *strlen(orig)*, pois a nossa string de destino tem que ter o caractere delimitador ao final.

Veja como fica nosso código em C:

```
char *strcpy(char *dest, char *orig)
{
    int i;

    for(i=0 ; i < strlen(orig) ; i++)
        dest[i] = orig[i];
```

```
    dest[i]='\0';

    return dest;
}
```

- **char *memcpy(char *dest, char *orig, int n) - Copia os 'n' primeiros termos da string *orig* para a *dest***

Primeiro recebemos do usuário duas strings: a 'orig', que é a string de origem, e a 'dest', que é a string de destino.

Ou seja, vamos pegar os 'n' primeiros caracteres da string de origem, e colocar nas 'n' primeiras posições da string de destino.

Como estamos passando strings (vetores), estamos passando seu endereço. Logo, essa função vai alterar a string de destino.

Para fazer isso, basta fazer com que a string *dest* receba os 'n' primeiros caracteres (da posição 0 até a n-1) da string *orig*.

Veja como ficou o código C para implementar essa função:

```
char *memcpy(char *dest, char *orig, int n)
{
    int i;

    for(i=0 ; i < n ; i++)
        dest[i] = orig[i];

    return dest;
}
```

- **int *memcmp(char *str1, char *str2, int n) - Diz se os 'n' primeiros termos de uma string é maior, menor ou igual aos 'n' primeiros termos da outra string**

Vamos comparar as 'n' primeiras posições das duas strings, ou seja, de 0 até n-1.
A comparação é feita caractere por caractere.

Lembrando que caracteres são, na verdade, números inteiros e sua representação está na tabela ASCII.

Assim, uma comparação de letras seria:
a = b
a < b
z > a

Portanto, podemos comparar diretamente os caracteres das strings.
Se em algum momento da comparação, algum caractere da string 1 for menor que o da string 2, a função pára e retorna -1.
Se em algum momento da comparação, algum caractere da string 2 for maior que o da string 2, a função pára e retorna 1.

Após todo o processo, se não retornar nem 1 ou -1, é porque as strings são idênticas e devem retornar valor 0.

Veja como fica nosso código:

```
int memcmp(char *str1, char *str2, int n)
{
    int i;

    for(i=0 ; i < n ; i++)
    {
            if(str1[i] < str2[i])
                return -1;
            else
                if(str1[i] > str2[i])
                    return 1;
    }
    return 0;
}
```

Note, porém, que uma letra maiúscula difere de uma minúscula. Para tratar essa comparação sem que isso importe, devemos passar os caracteres para maiúsculo ou minúsculo.

Para passar tudo para maiúsculo, podemos usar a função *toupper()*, da biblioteca *ctype.h*:

```
toupper( str1[i] ) < toupper( str2[i] )
```

Ou passar tudo para minúsculo, com a função *tolower()*, também da *ctype.h* e análoga ao exemplo acima.

Exercícios sobre Strings

00. Crie uma função que transforma todos os caracteres de uma string em maiúsculos.

01. Crie uma função que transforma todos os caracteres de uma string em minúsculos.

02. Crie uma função que recebe uma string e um caractere, e retorne o número de vezes que esse caractere aparece na string.

03. Crie uma função que recebe uma string e um caractere, e apague todas as ocorrências desses caractere na string.

04. Crie uma função que mostra os caracteres de uma string são repetidos.

05. Crie uma função que retira todas os caracteres repetidos de uma string.

Desafio 00: Strings Emo
Crie uma função que recebe uma string e transforma alguns dos caracteres em maiúsculos e outros em minúsculos. Faça sorteios com a função *rand()* para gerar números aleatórios em C, que serão usados para escolher os índices dos caracteres que serão alterados.

Desafio 01: Strings que gaguejam
Crie uma função que duplique cada caractere da string.

Structs

Já aprendemos um tipo básico de estrutura em C, os vetores.
Com eles, aprendemos a trabalhar com um número qualquer de variáveis do mesmo tipo.

A vantagem desse tipo de estrutura é não precisar trabalhar individualmente com cara variável.
Ao invés disso, trabalhamos com um grupo de variáveis.

Porém, trabalhamos sempre com variáveis de um mesmo tipo.
Em muitas aplicações de C, precisamos trabalhar com vários elementos de tipos diferentes, e é aí que entra o tipo de estrutura chamada *struct*, onde é possível agrupar os mais variados tipos de variáveis e formar um bloco de informação, que é a *struct*.

Como veremos, também, esse bloco de elementos pode ser considerado um novo tipo de dado.

Um tipo, assim como *int, float* ou *char*. Mas um tipo que você criou.

O que são, para que servem e onde são usadas as Structs

Agora que você estudou vetores e ponteiros, e aplicou esses conhecimentos nas strings em C, você viu a utilidade dos vetores, que é manusear várias variáveis de uma vez só.

Imagina a trabalheira de, ao invés de usar strings, usar variáveis do tipo *char* isoladamente?
Não dá nem pra pensar, não é?

Mas vetores têm uma limitação: eles possuem um tipo definido.
Ou o vetor é de char, ou de inteiros, ou de floats etc. Ou seja, vetores não podem ser usados com tipos diferentes.

Vamos ensinar agora as *structs*, que servem para manusear uma quantidade maior de informações de uma maneira mais simples, eficiente e de tipos diferentes.

- **Para que serve uma struct em C**

Vamos supor que você foi contratado por uma grande empresa para criar um aplicativo de C que armazena todas as informações dos funcionários, um banco de dados.

Como você faria para armazenar as idades deles?
Vamos supor que tem 300 funcionários.
Ora, você já sabe que não vai declarar inteiro por inteiro, então vamos declarar um vetor de idades:
int idades[300];

E os nomes? Vamos separar 50 caracteres para cada pessoa.
Temos que ter 300 vetores (strings), cada um com 50 caracteres. Ficaria:
char nomes[300][50];

O mesmo para sua data de admissão, o salário de cada um, seus números de identificação, horários, cargos etc.

Mas como você faria saber as informações de um funcionário especificamente?

Poderíamos pegar cada posição para uma pessoa. Por exemplo, o funcionário Bruce Dickinson seria o de número 50.

Ou seja, para pegar o nome dele, vá para a posição 49 da matriz de strings. Para saber o salário dele, vá na posição de número 49 do vetor de floats que representa os salários. Na posição 49 do vetor de inteiros podemos obter a idade deles e assim vai.
Isso tudo é possível, mas extremamente trabalhoso, confuso e nada organizado.

É para isso que existem as structs.

- **O que é uma struct em C**

Struct, ou estrutura, é um bloco que armazenam diversas informações. Poderíamos criar uma estrutura para cada funcionário, e dentro dessa estrutura tem variável do tipo int (para idade), do tipo string (para armazenar o nome), têm floats (que armazenam o salário de cada um), etc.

E cada estrutura teria um nome, que seria algo relacionado com o funcionário.
Assim, sempre que quiséssemos um dado de um funcionário, bastaríamos ter acesso a estrutura dele, e todas as informações viriam juntas.

Essas estruturas, ou structs, podem ter quantos elementos você queira e dos tipos que você quiser. Você primeiro define a estrutura, seu nome e elementos.
Depois você escolhe quantos elementos daquela estrutura quer ter, e os declara com nomes diferentes.

Após isso, em vez de você manipular as milhares de variáveis dos mais diversos tipos que tenha criado, você trabalha só com a estrutura.

Assim não tem que se ‘estressar’ com cada detalhe da estrutura. Esses detalhes são definidos somente uma vez, na declaração da estrutura.

- **Onde as structs em C são usadas**

Structs são muito usadas quando temos elementos em nossos programas que precisam e fazem uso de vários tipos de variáveis e características.

Usando *struct*, podemos trabalhar com vários tipos de informações de uma maneira mais fácil, rápida e organizada, uma vez que não temos que nos preocupar em declarar e decorar o nome de cada elemento da struct.

Por exemplo, vamos supor que você foi contratado para criar um aplicativo de uma escola.
As structs servem para você organizar as informações de uma maneira mais otimizada.
Para isso, basta colocarmos as informações comuns na estrutura.

Quais seriam os elementos comuns que deveríamos colocar nessa estrutura?

Ora, vamos trabalhar com alunos, então temos que colocar elementos na struct que representem os alunos: nome, notas, mensalidade, se esta mensalidade foi paga ou não etc.

Assim, podemos cria ruma struct para cada aluno e, automaticamente, esse aluno terá as variáveis acima citadas.

Outro exemplo seria para tratar carros de uma montadora.
Podemos representar a estrutura de um carro colocando alguns tipos de variáveis, que vão representar as características dos carros.

Por exemplo, qual a potência do motor, ano de fabricação, quantas portas possui, seu preço etc.

Assim, cada carro vai ter os elementos dessa struct automaticamente.

No próximo artigo de nosso curso de C vamos aprender como declarar uma struct, e você terá uma noção maior da utilidade da estrutura e como usá-las.

Como declarar uma struct em C

Agora que explicamos no tutorial passado sobre o que são as structs, para que servem e onde usá-las, vamos explicar **como** usar ditas cujas estruturas.

Nesse tutorial de C vamos ensinar você a declarar, preencher os elementos da struct, acessar e trabalhar com os elementos de qualquer struct.

- ## Como declarar uma struct em C

Como já foi dito, struct nada mais é que um conjunto, ou bloco, de variáveis. A sintaxe é a seguinte

```
struct Nome_de_sua_struct
{
        tipos nome_dos_tipos;
};
```

Vamos declarar, como exemplo, uma struct para representar os dados dos funcionários:

```
struct Funcionario
{
        int idade;
        char *nome;
        float salario;
};
```

- ## O tipo struct

Podemos ver as structs como um tipo de dado.
Por exemplo, “Funcionario”, que foi declarado anteriormente, pode ser visto como um novo de tipo de variável. É uma variável, ou um tipo, que define os funcionários.

Essa visão, de que criamos um novo tipo de variável, é tão certa que é possível criar e declarar mais variáveis do tipo “struct Funcionario”.
A sintaxe para declarar outras structs do tipo “struct Funcionario” é:

```
struct Funcionario empregado1;
struct Funcionario chefe;
struct Funcionario secretaria;
```

É aí que reside a beleza e importância das structs.
Note que no momento da criação do modelo da struct “Funcionario”, declaramos um inteiro, uma string e um float dentro da struct.
Assim, todas as structs do tipo “Funcionario” terão uma variável inteira, uma string e um float.

Sempre que declaramos uma nova struct desse tipo, essas variáveis são **automaticamente** criadas e terão essas variáveis. O que é uma verdadeira mão na roda.

Outra maneira de declarar variáveis de um tipo struct que queremos, é criar esses elementos após as chaves na hora de criar a struct Funcionario.
Veja:

```
struct Funcionario
{
        int idade;
        char *nome;
        float salario;
} empregado1, chefe, secretaria;
```

Pronto, agora existem 3 variáveis do tipo “struct Funcionario”: ‘empregado1’, ‘chefe’ e ‘secretaria’.

No próximo tutorial de nosso curso de C, vamos mostrar como acessar e alterar os elementos da struct, além de mostrar um exemplo através de um exercício resolvido e comentado.

Como acessar, alterar e ler os elementos de uma struct em C

Como já mostramos comodeclarar uma structe declarar variáveis do tipo struct que criamos, vamos aprender como acessar.

- **Como escrever e ler elementos de uma struct em C**

Vamos pegar nosso exemplo do tipo “struct Funcionario”.
Criamos 3 funcionários desse tipo, e todos eles terão os mesmos elementos internos, com os mesmos nomes: idade, nome e salario.

Se tem o mesmo nome, como vamos diferenciar esses elementos, então?
Pelo nome da variável do tipo “struct Funcionario”.

Para acessar o elemento ‘elemento’ da struct de nome “MinhaStruct”, usamos a sintaxe:
MinhaStruct.elemento

Apenas isso, basta botar um ponto após o nome que você escolheu para a struct.
Após isso, estamos acessando normalmente a variável:

chefe.idade-> é um inteiro como outro qualquer.
empregado1.nome-> é uma string como outra qualquer.
secretaria.salario-> é um float como outro qualquer.

Vamos mostrar, com o exemplo resolvido e comentado a seguir, que colocando o nome da struct, o ponto e o nome da variável, é como se tivéssemos acessando as variáveis de maneira normal, a única diferença é que essas variáveis estão dentro de uma struct.

- **Exemplo: Como declarar, preencher e acessar elementos de uma struct em C**

Defina uma struct para tratar de alunos. Dentro dessa struct, crie uma variável para armazenar o nome do aluno, e outras para armazenar as notas de Matemática, Física e a média dessas duas notas.
Após definir a struct, crie três variáveis do tipo struct que você criou.

Preencha os nomes e notas dos alunos, calculando automaticamente a média deles.
Depois exiba tudo isso.

Após definir a struct com o nome “Alunos” e seus elementos pedidos no enunciado da questão, vamos declarar 5 variáveis do tipo “struct Alunos”. Pode ser os nomes que você quiser, como: aluno1, aluno2, aluno3, aluno4, aluno5 ou qualquer outra coisa.

Mas você é um programador C, e é melhor que isso, estudou pelo curso C Progressivo, então você não pode perder tempo à toa. Vamos usar criar umvetorde elementos do tipo “struct Alunos”!

Vamos chamar esse vetor de ‘alunos’, assim teremos os elementos: alunos[0], alunos[1], ..., alunos[4]. Isso facilita pois podemos usar loopings para tratar essas variáveis, pois basta mudarmos o índice para mudar a variável do vetor.

O primeiro looping é para preencher os elementos das structs que representam os alunos.
Primeiro pedimos o nome, usando a função*gets()*, que recebe uma string como argumento.
No nosso caso, a string é:*alunos[índice].nome*
Isso porque declaramos a string de nome ‘*nome’*dentro da struct Alunos.

Os números:*alunos[índice].matematica*e*alunos[índice].fisica*, são floats como outros quaisquer que já trabalhamos, então podemos usar a nossa velha conhecidafunção*scanf()*.

E para calcular a média?
Ué,*alunos[índice].media*é um float como outro qualquer. Então somamos os floats que armazenam as notas de Matemática e Física de qualquer aluno, dividimos por 2 e temos a média de cada aluno.

Em seguida, apenas mostramos esses dados.
Veja como fica nosso código C:

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

```c
    struct Alunos
    {
        char nome[30];
        float matematica, fisica, media;
    };

    struct Alunos alunos[5];
    int count;

for(count = 0 ; count < 5 ; count++)
    {
        fflush(stdin);
        printf("\nNome do aluno %d: ", count+1);
        gets(alunos[count].nome);

        printf("Nota de matematica: ");
        scanf("%f", &alunos[count].matematica);

        printf("Nota de fisica: ");
        scanf("%f", &alunos[count].fisica);

        alunos[count].media = (alunos[count].matematica + alunos[count].fisica)/
2;
    }

    printf("\nExibindo nomes e medias:\n");

    for(count = 0 ; count < 5 ; count++)
    {
        printf("\nAluno %d\n", count+1);
        printf("Nome: %s\n",alunos[count].nome);
        printf("Media: %.2f\n", alunos[count].media);
    }

    return 0;
}
```

typedef: Como 'declarar' seus próprios tipos

Nos artigos passados, além de aprendermoscomo declarar uma struct na linguagem C, aprendemos aacessar e alterar os dados de uma estrutura.

Através de exemplos, vimos que podemos declarar várias variáveis de uma struct previamente declarada:
struct Aluno joaozinho;
struct Funcionario secretaria;

É como se tivéssemos criados tipos novos.
Assim como o 'int', 'float' ou 'char', é como se existisse o tipo 'struct Aluno' e 'struct Funcionario'.

Vamos entrar mais a fundo nesses detalhes e aprender como declarar structs através do atalho typedef.

- **O que é typedef**

A palavra reservada typedef nada mais é do que um atalho em C para que possamos nos referir a um determinado tipo existente com nomes sinônimos.

Por exemplo, com o typedef, em vez de termos que nos referir como 'struct Aluno', poderíamos usar somente 'Aluno' para criar structs daquele tipo.
Em vez de escrever sempre 'struct Funcionario', poderíamos escrever apenas 'Funcionario' e então declarar várias structs do tipo 'Funcionario'.

Embora possamos criar atalhos com typedef para outros tipos, o typedef é comumente usado com structs.

- **Como declarar tipos com typedef**

A sintaxe do typedef é bem simples:
typedef tipo_existente nome_que_voce_escolheu;

Onde 'tipo_existente' é um tipo já existente, como 'int' ou 'char'.
Agora, podemos passar a nos referir ao tipo 'tipo_existente' com outro nome, como 'nome_que_voce_escolheu'.

Exemplos de uso do typedef:

```
typedef int meuInteiro;
typedef char* String;
typedef unsigned short int idade;
```

Por exemplo, um aplicativo que cria atalhos para inteiro e ponteiro de char (vetor de caracteres):

```
#include <stdio.h>

int main(void)
{
    typedef int meuInteiro;
    typedef char String[20];

    meuInteiro numero = 1;
    String nome;
    scanf("%[^\n]s", nome);

    printf("A variavel do tipo 'meuInteiro' eh um int e vale %d\n", numero);
    printf("Ja a variavel 'nome' eh uma String e armazena \"%s\"\n", nome);

    return 0;
}
```

- **Como usar typedef para structs**

Como já havíamos dito no começo desse tutorial, o uso do atalho typedef é bem comum com structs, e a sintaxe é análoga ao que já foi apresentado, com alguns detalhes especiais que vamos explicar.

Quando formos usar o typedef com estruturas, não podemos declarar variáveis após o ponto-e-vírgula.
Colocamos apenas a palavra sinônima que vamos usar:

```
typedef struct Alunos
{
//código da
//struct
} Aluno;
```

Pronto, agora podemos fazer algo do tipo:

```
Aluno joao, maria;
```

Esse código faz com que possamos chamar "struct Alunos" apenas por 'Aluno'.
Note que 'Aluno' não é uma variável do tipo struct, é apenas um atalho, uma maneira mais simples de dizer 'struct Alunos'.

Outro exemplo de sintaxe de typedef com estruturas é a seguinte, onde não precisamos criar o nome da struct, apenas do sinônimo:

```
typedef struct
{
//código da
//struct
} Alunos;
```

Agora a podemos definir structs desse tipo apenas usando a palavra 'Alunos'.
Como exemplo do uso do typedef para deixar mais fácil a declaração de structs e outros tipos, vamos refazer o exemplo anterior com os conhecimentos aprendidos nesse tutorial de C.

- **Exemplo: Como declarar, preencher e acessar elementos de uma struct em C**

Defina uma struct para tratar de alunos. Dentro dessa struct, crie uma variável para armazenar o nome do aluno, e outras para armazenar as notas de Matemática, Física e a média dessas duas notas.

Após definir a struct, crie três variáveis do tipo struct que você criou. Preencha os nomes e notas dos alunos, calculando automaticamente a média deles.
Depois exiba tudo isso.

```
#include <stdio.h>
#define DIM 5
// Curso C Progressivo: www.cprogressivo.net
// Curso online e gratuito de C.
// Artigos, tutoriais e aulas sobre a linguagem C

int main(void)
{
    typedef struct
    {
        char nome[30];
```

```
        float matematica, fisica, media;
    }Alunos;

    Alunos alunos[DIM];
    int count;

    for(count = 0 ; count < DIM ; count++)
    {
        fflush(stdin);
        __fpurge(stdin);
        printf("\nNome do aluno %d: ", count+1);
        gets(alunos[count].nome);

        printf("Nota de matematica: ");
        scanf("%f", &alunos[count].matematica);

        printf("Nota de fisica: ");
        scanf("%f", &alunos[count].fisica);

        alunos[count].media = (alunos[count].matematica + alunos[count].fisica)/
2;
    }

    printf("\nExibindo nomes e medias:\n");

    for(count = 0 ; count < DIM ; count++)
    {
        printf("nAluno %d\n", count+1);
        printf("Nome: %s\n",alunos[count].nome);
        printf("Media: %.2f\n", alunos[count].media);
    }

    return 0;
}
```

Como enviar uma struct para uma função

Conforme você estudou pelos tutoriais sobre funções em C, sabe que para passar uma variável ou vetor para uma função, devemos declarar a função com o tipo de dado esperado.

Algumas semelhanças continuam, conforme veremos nesse artigo.

- **Como passar uma struct por argumento para uma função em C**

Assim como fizemos anteriormente ao longo de nosso curso, na hora de declarar a função declaramos o tipo de dado que ela vai receber (ou void, caso não receba nenhum argumento) e o tipo de dado que ela vai retornar (ou void).

E quando queremos passar uma struct, colamos o que, apenas *struct?*
Não faz sentido, pois o tamanho de uma struct em memória pode variar, dependendo da maneira como a criamos (depende de seus membros).

Se lembrar bem, no artigo passado vimos que podemos 'criar' (na verdade é criar um atalho) nossos próprios tipos com o uso do *typedef*.

Vamos criar, como exemplo, o tipo struct CARRO, que irá servir como padrão para declararmos alguns tipos de carros em uma concessionária.

```
typedef struct carro
{
        char modelo[30];
        float potenciaMotor;
        int anoFabricacao,
            numPortas;
} CARRO;
```

Ora, qual nosso tipo?
Pode ser tanto 'struct carro' como pode ser CARRO.

Assim, para declarar uma função que exibe os dados do carro para um cliente, teríamos que declará-la assim:
`void Exibe(struct carro car) {}`

Ou seja, 'car' é o parâmetro do tipo 'struct carro'.
Porém, usamos typedef, então em vez de 'struct carro', podemos nos referir a esse tipo de struct apenas por 'CARRO':
void Exibe(CARRO car) {}

- **Exemplo de código C: Passando uma struct para função**

Exercício: Defina uma struct de um carro. Crie um modelo de carro, preencha seus dados e exiba eles através de uma função que recebe esse tipo de struct.

Primeiro criamos a nossa struct, que será tipo CARRO, em seguida uma função que recebe um dado do tal tipo CARRO, que vamos chamar de ‘car’. Então, quando passamos uma struct do tipo CARRO para a função, ele é vista dentro da função como a struct de nome ‘car’, bastando acessar diretamente suas variáveis e exibi-las.

Dentro da **main()**, criamos uma variável, ou struct do tipo CARRO, de nome ‘fusca’.
A única coisa, talvez, diferente é a maneira com a qual inicializamos nossa struct: temos que preencher todos os termos da struct, na ordem.

Veja como ficou nosso código C

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Apostila online, tutorial completo sobre
// a linguagem de programação C !

typedef struct carro
{
    char modelo[30];
    float potenciaMotor;
    int anoFabricacao,
        numPortas;
}CARRO;

void Exibe(CARRO car)
{
    printf("Modelo: %s\n", car.modelo);
    printf("Motor: %.1f\n", car.potenciaMotor);
```

```
    printf("Ano: %d\n", car.anoFabricacao);
    printf("%d portas\n", car.numPortas);
}

int main(void)
{
    CARRO fusca = {"Fuscao preto", 1.5, 74, 2};
    Exibe(fusca);

    return 0;
}
```

Até aqui tudo ok, pois apenas *lemos* os membros de uma struct...se quisermos alterar os membros de uma estrutura, a coisa muda um pouco, conforme veremos no próximo tutorial.

Passando structs por referência para funções - O operador ->

Antes de mais nada, vamos deixar bem claro que, à rigor, não existe passagem por referência em **linguagem C**.
Mas por que vemos tanto falarem sobre isso, se não existe?

Na verdade, o que existe é uma espécie de 'truque', que é passar o endereço de memória, através de ponteiros, para funções, simulando uma passagem por referência.

Então não há problema em falar de passagem por referência em C, apenas use seu bom senso.

- **Como alterar uma struct em uma função**

Ok, ler os dados de uma struct através de funções é bem simples e é feito exatamente da mesma maneira que fizemos com outros tipos de dados.

Porém, só ler nem sempre é tão útil assim. E quando quisermos alterar uma struct?
Vamos ter que alterar na *main()* ?

Claro que não! Já explicamos que entupir a função *main()* é um péssimo hábito de programação. Temos que dividir bem as tarefas de um aplicativo em C em diversas funções, cada uma fazendo uma tarefa bem específica.

Se você lembrar bem, para alterar o valor de uma variável, fizemos uso da passagem por referência, onde enviamos o endereço de memória de uma variável, em vez de uma cópia de seu valor (como é feito na passagem por valor).

Para fazer, a função deve ser declarada de modo a esperar receber, como argumento, um endereço de memória. Ou seja, o parâmetro é um ponteiro. A nossa função preenche seria:
void Preenche(CARRO *car);

E para passarmos a struct ? Temos que passar o endereço de memória dessa estrutura, e isso é feito colocando o operador & antes do nome da struct:

```
Preenche(&fusca);
```

Até aí, tudo ok.
O problema vem quando vamos tratar a estrutura dentro da função, pois devemos ter um cuidado especial e é aqui onde os iniciantes costumam se confundir e se complicar.

A função vai receber um ponteiro ***car*** de uma estrutura.
Para alterarmos os membros da estrutura, temos que usar o asterisco antes do ponteiro, lembra? O ponteiro em si é só um endereço de memória, o valor para qual esse ponteiro aponta é obtido quando colocamos um asterisco antes do nome do ponteiro.

Por exemplo, para preenchermos o campo modelo:
```
gets( (*car).modelo);
```

E para acessar o elemento que armazena a potência do motor?
```
(*car).potenciaMotor
```

E para alterar esse valor, dentro da função *scanf()*, por exemplo?
```
scanf("%f", &(*car).potenciaMotor);
```

Então o código de nossa função que recebe e altera os membros de uma struct fica:
```
void Preenche(CARRO *car)
{
    printf("Modelo do carro: ");
    gets( (*car).modelo );

    printf("Motor: ");
    scanf("%f", &(*car).potenciaMotor);

    printf("Ano: ");
    scanf("%d", &(*car).anoFabricacao);

    printf("Numero de portas: ");
    scanf("%d", &(*car).numPortas);
}
```

- **O operador ->**

Embora tenhamos explicado passo por passo como chegar na função anterior, temos que assumir que fica um pouco confuso e complexo nosso código:
&(*car).numPortas

Note também que colocamos o asterisco e o ponteiro em parêntesis separado.
Consegue imaginar o motivo disso?

Vamos imaginar que em vez de: (*car).numPortas
Usássemos: *car.numPortas

Podemos interpretar isso de duas maneiras:
1. Estamos tentando acessar o elemento ‘numPortas’, que é um membro da struct que é apontada pelo ponteiro ‘car’. Que é o que vínhamos fazendo, usando parêntesis.
2. Estamos tentando acessar o ponteiro ‘numPortas’, da estrutura ‘car’.

Mas ‘car’ não é uma struct, muito menos tem o membro ‘numPortas’. Sabemos que ‘car’ é apenas um ponteiro pra uma struct, ou seja, ele é um endereço de memória. E é isso o que acontece quando não colocamos os parêntesis, pois para o C, o ponto final (.) tem procedência maior que o asterisco (*).

Para evitar problemas, usamos o parêntesis em volta do ponteiro ‘car’.
Mas ainda assim fica confuso, e em programação C, ficar confuso não é legal. Por isso, criaram um operador que vai facilitar isso, o ->

-> é simplesmente um atalho que substitui o asterisco e os parêntesis (é fácil esquecer e se confundir com estes), e com o ‘->’ a coisa se torna bem óbvia.
Por exemplo, em vez de: *(ponteiro).
Escrevemos: ponteiro->

No nosso exemplo do carro, as duas linhas seguintes são equivalentes:
scanf("%d", &(*car).anoFabricacao);
scanf("%d", &car->anoFabricacao);

Fica bem fácil de entender o: car -> anoFabricação

- **Exemplo: Alterando e exibindo structs através de funções em C**

Crie um programa na linguagem C que define a estrutura de um carro, altere seus dados através de uma função (use passagem por referência e o operador ->) bem como use outra função para exibir os membros da struct.

Nosso código, agora fazendo uso do operador -> , será:

```
#include <stdio.h>
// Curso C Progressivo: www.cprogessivo.net
// O melhor curso de C! Online e gratuito !
// Apostila online, tutorial completo sobre
// a linguagem de programação C !

typedef struct
{
    char modelo[30];
    float potenciaMotor;
    int anoFabricacao,
        numPortas;
}CARRO;

void Exibe(CARRO car)
{
    printf("\n\tExibindo carro\n");
    printf("Modelo: %s\n", car.modelo);
    printf("Motor: %.1f\n", car.potenciaMotor);
    printf("Ano: %dn", car.anoFabricacao);
    printf("%d portas\n", car.numPortas);
}

void Preenche(CARRO *car)
{
    printf("Modelo do carro: ");
    gets( car->modelo );

    printf("Motor: ");
    scanf("%f", &car->potenciaMotor);

    printf("Ano: ");
    scanf("%d", &car->anoFabricacao);

    printf("Numero de portas: ");
    scanf("%d", &car->numPortas);
```

```
}

int main(void)
{
    CARRO fusca;
    Preenche(&fusca);
    Exibe(fusca);

    return 0;
}
```

Bem mais fácil e óbvio de entender, não acham?

É importante entender bem o conceito do operador -> e da passagem de estruturas para funções, pois faremos uso desses conhecimentos mais a frente em nossa apostila de C, na seção de “Listas, Filhas e Pilhas”.

Exercícios sobre structs em Linguagem C

Usando os conhecimentos ensinados em nossa apostila de C, sobre:

- declaração, inicialização e acesso de structs
- criação de tipos com typedef
- como passar structs para funções
- uso do operador -> em estruturas

Resolva as seguintes questões:

00. Defina uma estrutura que irá representar bandas de música.
Essa estrutura deve ter o nome da banda, que tipo de música ela toca, o número de integrantes e em que posição do ranking essa banda está dentre as suas 5 bandas favoritas.

01. Crie um looping para preencher as 5 estruturas de bandas criadas no exemplo passado.
Após criar e preencher, exiba todas as informações das bandas/estruturas.
Não se esqueça de usar o operador -> para preencher os membros das structs.

02. Crie uma função que peça ao usuário um número de 1 até 5.
Em seguida, seu programa deve exibir informações da banda cuja posição no seu ranking é a que foi solicitada pelo usuário.

03. Crie uma função em C que peça ao usuário um tipo de música e exiba as bandas com esse tipo de música no seu ranking. Que função da string.h você usaria para comparar as strings que representam o tipo de banda?

04. Crie uma função que peça o nome de uma banda ao usuário e diga se ela está entre suas bandas favoritas ou não.

05. Agora junte tudo e crie uma mega aplicação em que exibe um menu com as opções de preencher as estruturas e todas as opções das questões passadas.

Alocação dinâmica de memória

Programar em C é, constantemente, se preocupar com coisas que você não se preocuparia se programasse na grande maioria das outras linguagens.

Quem nunca programou em C, C++, Assembly ou outra linguagem com acesso a memória, nunca se preocupou, ou sequer sabia que isso era necessário.

Geralmente as linguagens de alto nível já vem com um gerenciamento automático de memória.

Obviamente, esse gerenciamento nunca será perfeito, pois a máquina não pode adivinhar o que sua aplicação e sua cabeça quer que ela faça exatamente.

Se quiser ter um total controle sobre o uso de sua memória (inclusive otimizando e deixando suas aplicações as mais eficientes possíveis), você deve dominar bem todos os conceitos da alocação dinâmica de memória.

Com esses conhecimentos, você terá um controle total do tamanho de suas aplicações, bem como da eficiência e do uso racional de memória e processamento, assuntos constantemente ignorados por programadores de outras linguagens, mas essencial para quem quer se tornar um programador C.

O que é e para que serve a Alocação Dinâmica de memória em C

No decorrer de nossa apostila de C, frisamos várias vezes que a linguagem C costuma agir em baixo nível, ou seja, bem próximo ao hardware, na arquitetura de seu sistema.

É por isso que temos que declarar, manualmente, as variáveis, lidar com endereços de memória (ponteiros), buffer, ter cuidado para não extrapolar os limites de vetor e outros detalhes e preocupações que a maioria das linguagens não exigem do programador.

Nessa introdução, vamos explicar outra coisa que o programador C deve ter consciência: alocar somente o necessário de memória.

- **A alocação estática e seus problemas**

Alocar estaticamente memória é o que vínhamos fazendo até então, em nossa apostila completa de C online.

No começo de nossas aplicações sempre declaramos as variáveis, bem como o tamanho de vetores.

Por exemplo, para criar uma aplicação de um banco online, precisamos que o usuário cadastre seu nome.
Quantos caracteres você deixaria disponível para que ele fizesse isso? 30? 40?
50 já seria suficiente para a maioria das pessoas...mas e se fosse um descendente da família real brasileira?

Não é incomum encontrar pessoas com 5 ou 6 sobrenomes...então sua aplicação teria uma séria falha: o usuário iria digitar mais caracteres que o permitido.

Se você alocar 10 caracteres e escrever 20, vai ver que isso é possível. Porém extremamente falho e perigoso, pois os caracteres sobressalentes irão ocupar outros endereços de memória que você não alocou, e nesses espaços de memória poderiam existir informações importantes de seu computador.

É bem comum ouvirmos falar de ataques e vírus que agem assim, nessa falha de alocação de memória e processamento.

Provavelmente já deve ter ouvido falar em 'Stack overflow', que é quando um programa usa mais memória que o que foi pré-estabelecido, ou usa mais do que é esperado.

- **Quanto de memória devo usar?**

Vamos supor que você foi contratado por uma empresa para criar um aplicativo para gerenciar todos os funcionários.
A empresa tem 80 funcionários.

Você, como é esperto e visionário, vai criar uma struct para definir os funcionários e declarar logo 100 dessas estruturas, para armazenar os dados de seus funcionários.
Ok, sem problemas...você até garantiu espaço caso mais funcionários sejam contratados.

E se a empresa crescer demais? Passar dos 100 funcionários?
Mexer no código? Mudar pra quanto agora?

E se você decidir criar um programa de edição de textos, como o bloco de notas, quanto de memória você vai definir para uso?
1kb? Muito pouco.
1Mb? Razoável.
1Gb? Bastante, dificilmente alguém ia extrapolar isso num simples texto...
Mas há quem extrapole, e aí? Sua aplicação iria ficar limitar?

Mas 99% das pessoas só iriam usar alguns meros kb, e você iria alocar 1Gb de memória?
Que absurdo! Ia ser lento e ocupar o HD inteiro, ninguém iria usar seu programa.

Qual a solução?
Alocar o tanto que o usuário vai usar. E qual o tanto de memória que ele vai usar?
Ué, depende dele. Não é algo constante, não é estático. É dinâmico.

- **O que é alocação dinâmica de memória em linguagem C**

A maneira ideal de se trabalhar com memória é alocando somente o que vai se utilizar.
Óbvio, não?

Se a pessoa vai digitar 10 caracteres, armazene isso numa string de tamanho 11 (tem que ter o caractere limitador \0).

Se uma funcionária de uma farmácia vai cadastrar 20 medicamentos, seu aplicativo C deve alocar somente o espaço para as estruturas desses 20 medicamentes.

É isso que iremos aprender nessa seção do curso C Progressivo.
Sem dúvidas, é um dos tópicos mais importantes e diferenciados, pois são poucas as linguagens que vão permitir o programador ter acesso não só a memória, mas como e quanto da memória você vai usar.

Como diria o tio do Peter Parker, o Homem-Aranha: com grandes poderes, vêm grandes responsabilidades.

Como programador C, você tem total responsabilidade com o uso correto da memória.
Não use à toa, não use mais que o necessário, não gaste processamento quando não é necessário.

O diferencial da linguagem C é essa eficiência, se for para programar sem se preocupar com memória e processamento, não faz muito sentido programar em C.

Sabendo os conceitos e uso correto da alocação dinâmica de memória, vamos selecionar somente o tanto exato de memória que vamos usar, sem desperdiçar.

Isso vai fazer com que suas aplicações C fiquem bem menores do que já são, além de não sobrecarregarem sua máquina.

- **Onde a alocação dinâmica é usada**

Como ocorre com todos, você também irá cometer erros de programação, e errar quando estiver lidando com alocação dinâmica de memória, e isso irá trazer algumas consequências mais graves do que você está acostumado (geralmente não ocorre do jeito que você quer ou recebe um alerta do *debugger*) como lentidão e travamento do sistema.

Não é muito incomum vírus e aplicativos maliciosos fazerem uso excessivo de alocação de memória em seus programas maldosos.
Uma (das milhões) verificação que os anti-vírus fazem é checar quanto de memória seu sistema está usando.

Se esse número crescer muito e de forma estranha, certamente alguma aplicação está com erro ou está travando propositalmente seu sistema.

Um dos motivos da lentidão e travamentos de sistemas não muito otimizados, como Windows, é o mal gerenciamento da memória. Você abre um programa, ele aloca memória, usa, mas não libera corretamente essa memória após seu uso e usa mal durante a aplicação...então você já perde memória e processamento aí.

Depois você abre um jogo, usa absurdos de memória, que também é quase impossível de se gerenciar perfeitamente...lá vai mais memória e processamento para gerenciar essa memória...e assim vai indo, e seu sistema ficando lento, começando a travar, a demorar séculos para carregar algumas aplicações, e lá vai você gastar 50 reais para que o sobrinho da vazia venha formatar sua máquina, como se isso fosse algo normal (não, não é...experimente usar Linux).

Então, além de alocar, temos que desalocar (liberar) corretamente a memória quando não estamos mais usando.

Assim, os outros programas terão mais memória para usar.

Notou a importância da alocação dinâmica de memória em linguagem C?

A função malloc - Como alocar memória na linguagem C

No tutorial passado de nossa apostila de C, vimos a importância da alocação dinâmica da memória, técnica contrária a que vínhamos fazendo, que era declarando tudo estaticamente.

Vamos agora ensinar como usar a função malloc(), uma das três funções (malloc, calloc e realloc) que o C possui para tratar a alocação de memória, além de vermos, finalmente, na prática, uma grande utilidade dos ponteiros em C.

É uma importante lição que será extensivamente usada em nossas lições sobre listas encadeadas, filas e pilhas.

- **Como usar a função malloc da stdlib.h**

malloc(), de *Memory Allocation*, é uma função da biblioteca stdlib.h que recebe como argumento números inteiros positivos (size_t), que irão representar o número de bytes que iremos alocar.

Essa função retorna um ponteiro contendo o endereço do bloco alocado. Sua sintaxe é:

```
void *malloc(size_t numero_de_bytes);
```

Aqui, notamos uma peculiaridade da função malloc() (e da linguagem C). A função retorna um ponteiro, mas ao contrário do que tínhamos visto anteriormente, ela não retorna um ponteiro de tipo específico (int, float, struct etc).

Como estudamos bem, os ponteiros precisam saber para que tipo de variável vão apontar, pois (dentre outras coisas), podemos fazer operações matemática com ponteiros.

Por exemplo: ptr++;

Se esse ponteiro apontar para um caractere, ao incrementarmos, ele pulará uma posição de endereço de memória.

Já se apontar para um inteiro, ele pulará sizeof(int) posições de endereço, para apontar para o próximo inteiro de um vetor, por exemplo.
Por isso é importante, e essencial, que o ponteiro saiba pra que tipo de dado ele aponta.

Como a função malloc() serve para declarar qualquer tipo de dado, seja int, float, double ou uma struct criada por você, sua sintaxe foi mostrada como void.

Ela retorna o endereço do bloco de memória que foi alocado. Ao passo que fazemos essa alocação, devemos fazer um cast, ou seja, fazer com que um ponteiro (de algum tipo já definido) receba essa endereço.
Se quisermos alocar um bloco de endereços para inteiros, ao invés do void* colocamos:

```
(int *) malloc(size_t bytes);
```

Lembrando que a função retorna um endereço de memória.
Logo, alguém (ponteiro) deve receber esse retorno.

Por exemplo, se quiséssemos alocar 20 caracteres para conter uma string, devemos fazer:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *nome;
    nome = (char *) malloc(21);

    printf("Digite seu nome: ");
    gets(nome);

    printf("%sn", nome);

    return 0;
}
```

(Por que alocamos 21 caracteres, se queremos usar apenas 20?)
Poderíamos criar o ponteiro e logo na declaração fazer ele receber o endereço de um bloco alocado de memória:

```
char *nome = (char *) malloc(21);
```

- **Dica: usar sizeof()**

Outra boa prática é evitar o uso de números para escolher o número de bytes alocados.

Isso se deve ao fato de diferentes variáveis terem diferentes valores, dependendo da arquitetura.

Há máquina que um inteiro ocupa 2 bytes, em outras ocupam 4 bytes.

Sim, é um processo trabalhoso e enfadonho. E isso não é o pior.
Como veremos no próximo tutorial de C de nossa apostila, vamos ver os principais problemas decorrentes de uma má alocação de memória.

É um assunto um pouco delicado, mas não devemos fugir do uso dinâmico de memória, pois essa é uma das técnicas mais úteis para se otimizar programas, deixá-los mais rápidos e fazer menos uso de memória.

Para alocar de maneira correta, sem medo de errar (e de ser feliz), use a função sizeof():

```
char *nome = (char *) malloc(21*sizeof(char));
```

Sim, é um processo trabalhoso e enfadonho. E isso não é o pior.
Como veremos no próximo tutorial de C de nossa apostila, vamos ver os principais problemas decorrentes de uma má alocação de memória.

Por hora, estamos mostrando apenas exemplos simples.
Obviamente que os computadores atuais tem muitos mais memória que 20 bytes para um nome, e em aplicativos simples como este, a alocação não é necessária.

Mas para trabalhar com microprocessadores e microcontroladores, por exemplo, onde o tamanho da memória é algo crítico, é necessário ter controle de cada byte.

Mas ainda nesse curso você verá uma utilização bem prática e essencial da alocação dinâmica de memória (e consequentemente, mais uma aplicação de ponteiros), que será no estudo de estruturas dinâmicas de dados, para fazer Listas encadeadas, Filas e Pilhas.

Por hora, vamos treinar os conceitos básicos.

- **Exercício: Calculando a média de qualquer quantidade de números com malloc()**

Crie um programa que calcula a média de uma quantidade qualquer (informada pelo usuário) de números.

O programa deve armazenas esses números em um vetor. Depois, use esse vetor para mostrar todos os números e mostrar a média dele.

Use alocação dinâmica de memória para colocar os números no vetor. Não desperdice memória.

Resolveremos essa questão no próximo tutorial de nossa apostila de C, quando ensinarmos a desalocar (liberar) a memória alocada, com a função **free()**.

A função free() - Liberar e evitar vazamento de memória
No artigo passado de nossa apostila de C, ensinamos o como alocar memória de uma maneira dinâmica, através da função malloc() da biblioteca stdlib.h

Agora vamos ensinar como liberar essa memória que foi previamente alocada, que é um bom hábito que evita um famoso problema, o vazamento de memória. E ao término do tutorial, vamos resolver um exercício que foi proposto no artigo passado, onde iremos mostrar o uso das funções malloc() e free().

Memory leak (vazamento de memória) em linguagem C

No artigo inicial desta seção sobre alocação dinâmica de memória, citamos vários motivos pela qual o uso desta técnicas é imprescindível para qualquer programador C, e é um verdadeiro diferencial em relação à grande maioria das outras linguagens de programação, pois poucas irão lhe propiciar o poder de controlar cada byte da memória de seu computador.

Vimos que ao usar a função malloc() estamos, na verdade, reservando um espaço em memória. Isto se chama alocar, e uma vez feito isso, aquela região da memória estará protegida e não será possível usar ela para outro propósito.
Porém, por mais que tenha uma memória grande, ela será sempre limitada, e há certas aplicações (como um sistema operacional ou um jogo de alta performance) que irão exigir, e muito, de sua memória.

E é aí que surge o problema que embasa este artigo de nossa apostila: se você alocar muita memória, chegará uma hora que não vai ter mais nenhum byte disponível para uso e seu software vai simplesmente parar (as vezes seu computador simplesmente trava e não volta nem com reza brava).

Geralmente isso ocorre por uma falha de programação, pois o programador simplesmente deixou muita memória 'vazar'.
Uma maneira simples de acontecer isso é ficar alocando memória dentro de um looping e não liberar ela antes do término desse looping.

Como você deve ter estudado funções em C, quando criamos uma variável dentro do escopo da função, é como se ela não existisse fora dela. Então podemos criar um ponteiro, alocar memória e fazer o dito cujo ponteiro armazenar o endereço da memória alocada. Se não liberarmos a memória antes da função acabar, ela vai ficar eternamente alocada e inacessível, pois 'perdemos' o ponteiro quando a função terminou.

Vamos mostrar como é isso na prática.
Criamos uma função chamada aloca(), que não recebe nem retorna nada.
Dentro dela criamos um ponteiro para um inteiro e em seguida alocamos 100 bytes, através da malloc().
Obviamente, quando a função termina, esse ponteiro deixa de existir, mas a memória reservada continua lá, e inacessível.

Na nossa main() fazemos infinitas chamadas da função aloca(), que aos poucos vai reservando de 100 em 100 bytes a memória de seu computador. E aos poucos você vai notar sua máquina ficando lenta, pois vai haver cada vez menos memória, vai ficando cada vez mais 'difícil' de achar espaços para alocar até...sua máquina travar completamente.

Veja, é um código bem simples, usa coisas básicas que aprendemos aqui em nosso curso e mostra bem o que é o vazamento de memória:

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

void aloca()
{
 int *ptr;
 ptr = (int *) malloc(100);
}

int main(void)
{
 while(1)
  aloca();

 return 0;
}
```

Se estiver no Linux, abra uma janela de seu terminal e digite: free -m -s 1
Esse comando irá mostrar, a cada 1 segundo, o estado das memórias livres em sua máquina.

free(): A função que libera memória

A solução para este tipo de problema é simples, basta usar a função free(), que vai liberar o espaço de memória que foi previamente alocado.

Assim como outras funções de alocação dinâmica de memória, esta função também está na biblioteca stdlib.h

Ela recebe um ponteiro, o que foi usado para receber o endereço do bloco de memória alocada, e não retorna nada.
Ou seja, sua sintaxe é bem simples:

free(ponteiro);

O grande problema reside na pergunta 'Onde liberar memória?'

Se seu projeto for mais simples, provavelmente só vai precisar liberar ao final de sua aplicação (embora a memória seja geralmente liberada após terminar um programa, é uma boa prática de programação sempre liberar o que foi alocado antes de sua aplicação terminar).

Mas, basicamente, devemos liberar a memória sempre que não formos mais usar o que foi alocado.

Isso geralmente acontece ao final das funções ou loopings que fazem pedidos de alocação de memória.

Como exemplo, podemos 'consertar' o exemplo de código passado, simplesmente colocando free(ptr) ao final da função.

Veja que agora ela pode rodar infinitamente, pois a memória alocada é liberada após seu 'uso'.

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

void aloca()
{
 int *ptr;
 ptr = (int *) malloc(100);
```

```
 free(ptr);
}

int main(void)
{
 while(1)
  aloca();

 return 0;
}
```

- **A importância do bom gerenciamento de memória**

Como havíamos dito, usar alocação de memória fará sempre seus programas serem mais robustos e seguros, por isso indicamos que use sempre em seus projetos, mesmo nos mais simples.

E obviamente, sempre libere a memória alocada após seu uso, pois o vazamento de memória é um problema muito comum, que certamente você irá se deparar, caso siga a profissão.

É um problema tão habitual, que atormenta tanto os programadores, que existem até ferramentas para detectar os 'leaks memory'. Muitas vezes isto ocorre não por um erro, mas sim por um ataque.

Os anti-vírus, por exemplo, estão sempre gerenciando o uso da memória dos computadores que protegem, pois é um tipo de ataque comum, consumir toda a memória de um sistema, para travá-lo e tirá-lo do ar.

Se isso pode ocorrer em máquinas boas e modernas, imagine para quem trabalha com microcontroladores, por exemplo, que muitas vezes possuem apenas alguns poucos kilobytes de memória.

Não é à toa que engenheiros de computação que trabalham com hardware estão sempre preocupados com a maneira na qual usam a memória de seus projetos.

Linguagens de programação de alto nível, como o Java, geralmente fazem de maneira 'automática' esse gerenciamento de memória. Geralmente funciona de maneira razoável, raramente funciona de maneira perfeita.

O ideal, para garantir que seu aplicativo seja sempre o mais confiável, robusto, seguro e rápido, só mesmo gerenciando 'na mão'.

Como diz o ditado: Se quer algo bem feito, faça-o você mesmo.

- **Alocação de memória, free() e segurança**

Embora o ensino da alocação e liberação de memória seja comum em vários livros e cursos, algumas coisas passam batido, principalmente no que se refere à segurança de uma aplicação.

Vamos mostrar agora uma maneira bem comum de explorar falhas através dos ponteiros.
Vamos criar um programa simples, que irá pedir uma senha e armazenar num local que foi previamente alocado.

Então você usa essa senha como quiser, e como é um bom programador, irá usar a free() para liberar a memória que foi usada, até mesmo por questões de segurança.

Porém, ao contrário do que muitos pensam (e aí que mora o perigo), ao liberar a memória você não vai apagar os dados existentes nela, você vai apenas dizer ao seu computador "Hey, esse bloco de bytes aqui, já usei, então você pode pegar para fazer outra coisa".

Mas as informações ainda estão lá. E como obter o que tem lá?
Através do ponteiro que ainda aponta para lá. Ou seja, a free() não vai mudar o endereço armazenado no ponteiro, ele ainda vai continuar apontando para sua senha mesmo após aquele bloco de memória ter sido liberado.

Veja o código do programa:

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


int main(void)
{
 char *senha;
```

```
 senha = (char *) malloc(21*sizeof(char));
 printf("Digite sua senha [ate 20 caracteres]: ");
 scanf("%[^\n]s", senha);

 printf("Senha: %s\n", senha);
 printf("Endereço antes da free(): %d\n", &senha);

 free(senha);

 printf("Endereço depois da free(): %d\n", &senha);

 return 0;
}
```

O resultado:

```
Digite sua senha [ate 10 caracteres]: CProgressivo
Senha: CProgressivo
Endereço antes da free(): 1993476312
Endereço depois da free(): 1993476312
```

Ou seja, o ponteiro maroto continua apontando pro local da memória que está minha senha, e alguém poderá usar de uma maneira bem maléfica caso eu não tenha gravado outra coisa por cima desse bloco de memória. Como nos proteger, então?

Uma boa prática de segurança é que sempre que for 'liberar' seus ponteiros, fazer eles apontarem para NULL.
Ou seja, faça:
ptr=NULL;

Sempre que usar a free(), pois embora tenha liberado a memória para outro uso, o ponteiro continuará apontando para aquele endereço de memória. Assim, o ponteiro não vai mais te dedurar.

- **Exercício resolvido: Usando malloc() e free()**

No tutorial passado de nosso curso, passamos o seguinte exercício:
Crie um programa que calcula a média de uma quantidade qualquer (informada pelo usuário) de números.
O programa deve armazenas esses números em um vetor. Depois, use esse vetor para mostrar todos os números e mostrar a média dele.

Use alocação dinâmica de memória para colocar os números no vetor. Não desperdice memória.

Vamos resolvê-lo agora para ilustrar o uso da malloc() e da free().
Na main(), o programa inicia um looping, que só para se o usuário digitar 0.

Neste looping é pedido um número inteiro, que será o tanto de números que o usuário vai digitar.
Após ele fornecer essa informação, passamos ela para a função aloca() que vai alocar dinamicamente um vetor de inteiros, com o número de elementos exato que o usuário digitou, e retornar o endereço desse espaço alocado.

Esse endereço é armazenado no ponteiro *numeros, da main().
Em seguida, mandamos esse vetor e o número de elementos para a função media(), que irá calcular a média de todos os elementos deste vetor e retornar esse float (a média de inteiros pode ser um número decimal).

Também mandamos os mesmos argumentos para a função exibe(), que irá mostrar os números digitados.

Após cada iteração do laço while, devemos liberar a memória que está apontado pelo ponteiro *numeros, senão fizermos isso a função aloca() vai alocar um espaço diferente de memória a cada iteração, consumindo a memória aos poucos.

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

int *aloca(int num)
{
 int count,
  *numbers;

 numbers = (int *)malloc(num*sizeof(int));

 for(count=0 ; count < num ; count++)
 {
  printf("Numero [%d]: ", count+1);
  scanf("%d", &numbers[count]);
 }
```

```c
 return numbers;
}

float media(int *numbers, int num)
{
 float media=0.0;
 int count;

 for(count=0 ; count<num ; count++)
  media += numbers[count];

 return media/num;
}

void exibe(int *numbers, int num)
{
 int count;

 for(count=0 ; count < num ; count++)
  printf("%3d", numbers[count]);
}

int main(void)
{
 int num=1,
  *numeros;

 while(num)
 {
  printf("Media de quantos numeros [0 para sair]: ");
  scanf("%d", &num);

  if(num > 0)
  {
   numeros = aloca(num);
   exibe(numeros,num);
   printf("\nA media destes numeros eh: %.2f\n", media(numeros, num) );
   free(numeros);
  }
 }

 return 0;
}
```

Esse programa calcula a média de 1, 2, 10, mil ou 1 milhão de números. E o melhor, só aloca 1, 2, 10, mil ou exatos 1 milhão de bytes, nem um a mais.

Extramente econômico, não deixa seu computador 'lerdo' por consumir memória demais, o que é um grande problema ocasionado por péssimos programadores.

A Wikipedia possui um excelente artigo sobre Memory Leak.

A função realloc(): realocando memória dinamicamente e a calloc()

Neste tutorial de nossa apostila de C, iremos aprender o que é a função realloc(), para que serve o realocamento de memória, vamos ver como fazer isso através de exemplos de códigos comentados, alar sobre a função calloc(), além de dar mais dicas sobre alocação dinâmica de memória.

Esta função completa nosso estudo sobre a alocação dinâmica, junto com os artigos sobre a função malloc() e sobre a liberação de memória com a função free().

- **Problemas com a malloc()**

Quando ensinamos o uso da função malloc() explicamos que ela aloca um determinado número de blocos 'n' de tamanho 'tam' bytes cada, e retorna o endereço desse bloco, através da seguinte sintaxe:
malloc(n*tam);

Aparentemente, ela está ok.
Mas dependendo da situação que estejamos analisando, a função malloc() pode não ser o suficiente para não resolver alguns problemas. Na verdade, veremos problemas que ela não vai sempre resolver da maneira mais eficiente possível.
Isso decorre do fato de que temos que saber o 'tanto' de blocos que vamos alocar.

Vamos supor que precisemos de 10 blocos de memória do tipo int.
Depois, descobrimos que precisamos de mais 2. O que fazemos?
Usamos a malloc para armazenar 12 blocos, passamos o conteúdo alocado previamente e liberamos o espaço anterior.

Resolvemos uma questão no artigo passado, que calculava a média de qualquer quantidade de números fornecida pelo usuário.
Porém, lá alocávamos o tanto certo, calculávamos a média e depois liberávamos o bloco de memória.
Ou seja, não precisávamos do que tinha sido alocado antes, simplesmente alocávamos outro bloco de memória, para cada operação.

E vamos parar para pensar nesse exemplo.

O usuário vai pedir a média de 10 números. Ok, você aloca facilmente os 10 com o uso da malloc().
Na próxima iteração, ele pede a média de 12 números, então você libera o anterior e aloca outro bloco.

Cá entre nós, não seria mais fácil pro computador simplesmente alocar mais 2 espaços? Quem sabe existe mais 2 blocos de memória ali, logo ao lado daqueles 10.

E ainda falando no problema que citamos.
Suponha que você alocou um espaço para armazenar 1000 structs, com dados sobre 1000 alunos de uma escola.
Ok, até aí tudo bem.

Mas no ano seguinte entraram mais 200 alunos. E aí?
Aloca um bloco de 1200 e copia o conteúdo anterior, não é?
Não seria mais fácil, rápido e eficiente se pudéssemos simples alocar mais 200 blocos, sem alterar os 1000 anteriores?

Seria melhor ainda se pudéssemos alocar esses 200 blocos ao lado dos 1000 previamente alocados.

Pois são esses, e outros tipos de contratempos, que a função realloc() vai resolver.

- **A função realloc(): O que é, para que serve e como usar**

Agora que já mostramos que nem sempre a malloc() é a solução mais eficiente e produtiva, vamos mostrar como tais problemas podem ser contornados, através do uso da função realloc().

Como o próprio nome diz, ela realoca um espaço de memória.

Ou seja, para realocar é necessário que algo tenha sido alocado. Então, antes de ver a sintaxe da realloc() podemos concluir que para usar ela é necessário ter um ponteiro que foi usado para alocar um espaço de memória.
A realloc(), assim como a malloc(), retorna um endereço com um novo bloco de memória.

Seja 'ptr' esse ponteiro, a sintaxe para o uso da função realloc() é:
realloc(ptr, numero_bytes);

O 'numero_bytes' é o número de bytes que queremos realocar. É exatamente da mesma maneira que fizemos como na malloc(), onde geralmente fazemos 'n * sizeof(tipo)'.

Outra coisa que devemos lembrar é que um ponteiro deve receber o endereço da realloc(), ou seja, não devemos usar ela de maneira 'solta', alguém devem receber seu retorno. É um erro comum simplesmente escrever algo do tipo:
realloc(ponteiro, n*sizeof(int))

E achar que agora o ponteiro 'ponteiro' foi usado para alocar aquele espaço de memória, quando não foi.
Se quisermos fazer isso, devemos capturar o retorno:
ponteiro =(int *) realloc(ponteiro, n*sizeof(int));

Note que devemos usar o cast de ponteiros aqui também, como na malloc.
Ou seja, se o ponteiro 'ptr' aponta para float, fazemos:
ptr = (float *) realloc(ptr, n*sizeof(float));
Se for do tipo char:
ptr = (char *) realloc(ptr, n*sizeof(char));

- **Memória não alocada**

Antes de mostrar um exemplo prático do uso da função realloc(), vamos fazer uma pausa neste tutorial de nossa apostila para dar mais uma dica, um bom hábito de programação que você deve ter.

Até o momento estávamos agindo como se a memória fosse sempre alocada, o que geralmente ocorre, pois as máquinas atuais possuem muito espaço em memória, e para as aplicações simples e básicas de nosso curso, precisamos de alocar pouca memória.

Porém quando você se tornar profissional e for criar aplicativos mais complexos e robustos, ou for trabalhar com dispositivos com pouca memória

(como microcontroladores), verá que nem sempre existe espaço suficiente de memória.

Quando pedimos memória e não há espaço suficiente, a função retorna o endereço NULL.
Por isso, faça sempre um teste condicional IF após alocação de memória, para tratar o caso em que não exista espaço suficiente de memória.

Como exemplo, vamos tentar alocar um espaço de memória absurdamente grande, e vemos a mensagem de erro:

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

int main(void)
{
 int* ptr = (int *) malloc(10000000000000000000);

 if(ptr == NULL)
  printf("Sem espaço suficiente\n");

 return 0;
}
```

A realloc() exige que você envie um ponteiro para ela por que pode ser que o novo bloco de memória que você pediu não esteja adjacente ao bloco que você indicou através do ponteiro, então o computador vai buscar outro bloco de endereço, daí o endereço do ponteiro muda.

Esse ponteiro que você passou, não necessariamente tinha que ter sido usado para aloca memória, pois se ele apontar para NULL, por exemplo, a realloc() vai funcionar exatamente como a malloc().

- **Exemplo de código: Como usar a função realloc() em C**

Crie um programa que armazene dinamicamente números fornecidos pelo usuário.
O programa deve perguntar quantos números o usuário quer adicionar e receber tais números.

Não desperdice memória e tempo, use a função realloc() para realocar memória sempre que o usuário quiser inserir mais números.

Vamos usar 3 variáveis.
Um inteiro 'opcao' para armazenar a opção do usuário no menu, outro inteiro 'size' que vai armazenar o tamanho do array de números e o ponteiro para inteiro, que irá receber o endereço do array alocado.

Na função menu(), simplesmente colocamos as opções de Sair, Colocar mais números ou Exibir a lista de números.
Essa função retorna o inteiro da escolha do usuário.

Vamos usar o retorno dessa função menu() dentro de um teste condicional switch que irá tratar cada opção descrita pelo usuário.
Esse switch está dentro de um laço do while, que só termina quando o usuário digitar 0 (opcao=0).

A primeira função é realoca(), que vai receber dois argumentos: o ponteiro que irá guardar o endereço do bloco alocado e o tamanho do array de inteiros. Porém, não vamos enviar o inteiro 'size', e sim seu endereço de memória, pois queremos que este inteiro seja alterado dentro da função realoca().

Dentro desta função nós perguntamos quantos números o usuário quer adicionar à lista, e armazenamos essa informação na variável 'add'.
Agora vamos realocar nosso bloco de memória, que antes era do tamanho '*size' (lembre-se que passamos o endereço de memória do inteiro, então para pegar o valor armazenado nesse endereço usamos *size em vez de size, como ensinamos em nossas seção sobre Ponteiros em C de nossa apostila).

Depois vamos pedir esses números para os usuários, e vamos adicionar os números no array. Fazemos isso através da variável 'count' dentro de um laço for. O count vai de 0 até (add-1). Então esses novos números vão sempre no final do array, a partir da posição (*size) até a (*size + add - 1).

Feito isso, temos que mudar o valor da 'size', pois nosso array cresce. Era *size, agora é (*size + add), fazemos isso assim: *size += add

E feito, retornamos o novo ponteiro, com os novos números adicionados à lista.

Vale lembrar que após usar a realloc() devemos checar se seu computador conseguiu espaço em memória.
Se conseguiu, o teste if(ptr) retorna valor lógico TRUE, pois 'ptr' não é NULL, e fazemos a alocação.

Caso não tenha conseguido alocar o espaço de memória, o ponteiro 'ptr' vai apontar para NULL e o teste será falso, indo para o else que termina o programa.

Por fim, a função exib() recebe o ponteiro e o tamanho do array, e simplesmente exibe todos seus elementos.
Sem segredo.
E, como não podemos esquecer, liberamos a memória alocada, que é pontada pelo ponteiro 'ptr', através da função free().

Logo, o código de nosso programa em C fica:

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

int* realoca(int* ptr, int* size)
{
 int count,
  add;

 printf("Deseja adicionar quantos numeros: ");
 scanf("%d", &add);

 ptr = (int *) realloc(ptr, (*size + add)*sizeof(int) );
 if(ptr)
 {

  for(count=0 ; count < add ; count++)
  {
   printf("Numero [%d]: ", count+1);
   scanf("%d", &ptr[*size + count]);
  }

  *size += add;
```

```
 }
 else
 {
  printf("Espaço em memória insuficiente\n");
  free(ptr);
  exit(1);
 }
 return ptr;
}

void exibe(int* ptr, int size)
{
 int count;
 for(count=0 ; count<size ; count++)
  printf("%3d", ptr[count]);

}

int menu()
{
 int opcao;

 printf("\nO que deseja: \n");
 printf("0. Sair\n");
 printf("1. Adicionar numeros\n");
 printf("2. Exibir lista de numeros\n");
 printf("Opcao: ");
 scanf("%d", &opcao);

 return opcao;
}

int main(void)
{
 int opcao,
  size=0,
  *ptr=NULL;

 do
 {
  switch(menu())
  {
   case 0:
     opcao=0;
     break;
```

```
   case 1:
     ptr=realoca(ptr, &size);
     break;

   case 2:
     exibe(ptr, size);
     break;

   default:
     printf("Opcao invalida!\n");

  }
 }while(opcao);

 free(ptr);

 return 0;
}
```

- **A função calloc()**

Para finalizar nossos estudos sobre alocação dinâmica de memória, vamos falar da função calloc() que é parecidíssima com a malloc(). Uma dessas diferenças é na sintaxe, porém seu propósito é o mesmo: alocar blocos de bytes em memória.

A sintaxe da calloc() é:
calloc(numero, tamanho_em_bytes);

Lembre-se que na malloc fazíamos: malloc(numero * tamanho_em_bytes)
Também usamos casting nos ponteiros.

Por exemplo, para alocar 'n' blocos de inteiros, com a calloc fazemos:
ptr = (int *) calloc(n, sizeof(int));

O mesmo para float:
ptr = (float *) calloc(n, sizeof(float));

- **Diferença entre calloc() e malloc()**

Se pensar um pouco, a função calloc() é praticamente igual à malloc(). Porém, há uma pequena diferença.

Quando usamos a malloc() simplesmente reservamos um espaço de memória.
Já quando usamos a calloc(), além de reservar esse espaço de memória ele muda os valores contidos nesses bytes, colocando todos para 0. É como se usássemos a função memset(), que inicializa um bloco de memória com valor 0.

Então, quando devemos usar malloc() e calloc()?
Se você tiver que inicializar um bloco de memória com 0, faça isso usando a calloc(), pois é mais simples e otimizado que fazendo isso manualmente. Na verdade, enquanto seu computador está ocioso ele faz com que alguns espaços de memória recebam esse valor 0, e quando você for usar a calloc() ele vai procurar um bloco que esteja 'zerado', então, no geral, a calloc() é mais rápida e otimizada que a malloc().

A calloc() é muito usada para se trabalhar com vetores multidimensionais (matrizes), pois facilita para alocar uma certa quantidade de números de vetores, por exemplo.

Outro uso é no quesito segurança.
No tutorial passado de nosso curso nós mostramos que após usar um bloco de memória devemos apontar o ponteiro para NULL, pois senão ele continua apontando para o local antigo da memória, e isso seria uma grave falha, um brecha no sistema.
A vantagem da calloc() é que ele apaga os dados que existiam antes naquele bloco de memória, fazendo assim uma segurança maior, pois apaga as informações anteriores.

- **Exercícios propostos com calloc() e realloc()**

1. Defina uma função chamada callocc(), que faz exatamente o que a calloc() faz.
Faça ela usando as funções malloc() e a memset(), que recebe três argumentos (o ponteiro, o que queremos colocar em todas as posições do vetor e o número de bytes):

memset(ptr, '\0', numero * tamanho_em_bytes);

2. Crie um programa que forneça os números de Fibonacci, o usuário escolhe o n-ésimo termo, e você fornece.
Calcule usando a realloc() para alocar memória.

Crie uma opção de modo que o usuário pode pedir outro termo, e esse novo termo deve ser achado da maneira mais eficiente possível (sem ter que recalcula todos os elementos de novo, pois os elementos da consulta anterior ainda estão no vetor - lembre-se que a realloc() aloca novos bytes e não apaga os anteriores).

O termo 't(n)' da sequência de Fibonacci é a soma dos dois anteriores
t(n) = t(n-1) + t(n-2), onde t(0)=0 e t(1)=1.

A sequência de Fibonacci revela coisas interessantíssimas sobre diversos fatos da natureza:

http://pt.wikipedia.org/wiki/N%C3%BAmero_de_Fibonacci

Estruturas de dados dinâmicas

Listas, Filas, Pilhas e Árvores

Através desse artigo de nossa apostila de C, iniciamos um dos mais importantes estudos em linguagem C, onde iremos aprender os conceitos de Listas, Filas, Pilhas e Árvores.

São elementos das estruturas dinâmicas de dados e são essenciais na linguagem C e no estudo da computação, de um modo geral.
Aprendendo estrutura de dados dinâmica, iremos nos aproximar do conhecimento que é realmente empregado nos mais diversos aplicativos e sistemas existentes.

- **O que é Estrutura Dinâmica de Dados em Linguagem C**

Aqui em nossa apostila de C, já estudamos estrutura de dados quando aprendemos sobre vetores, que foi uma maneira de manipular várias informações (variáveis ou estruturas) de uma vez só, pois poderíamos declarar vários desses elementos.

Quando isso era feito, esses dados estavam em uma sequência rigidamente definida na memória, bem como o número de elementos eram constantes. Esses dois detalhes, na verdade, possuem algumas limitações.

Ou seja, estamos um trecho do título deste tutorial, a parte de "Estrutura de Dados".
Também estudamos a outra parte, que se refere ao dinamismo. Vimos isso em nossa seção de Alocação dinâmica de memória, onde deixamos de estar presos ao conceito de "número de elementos fixos", pois lá alocávamos e realocávamos o tanto de espaços de memória que queríamos, da maneira que queríamos.

Pois bem, agora é hora de 'unir forças' e usar esses poderosos conceitos para criar outro importante conceito, o de estrutura de dados dinâmicas, ou seja, as estrutura de dados que são alocadas, realocadas e movidas o quanto e do jeito que quisermos.

Ou seja, ao contrário de vetores de tamanhos fixos, iremos basicamente colocar estruturas(structs) em uma sequência que será criada

dinamicamente, da maneira que quisermos. Ou seja, podemos colocar elementos nessa sequência, tirar, mudar de lugar, percorrer e fazer coisas incríveis.

Mas só falando dessa maneira, a coisa fica um pouco confusa e abstrata. Vamos entrar em detalhes nos tipos de estrutura de dados que iremos estudar: Listas, Filas, Pilhas e Árvores.

- **O que é uma Lista em C (Lists)**

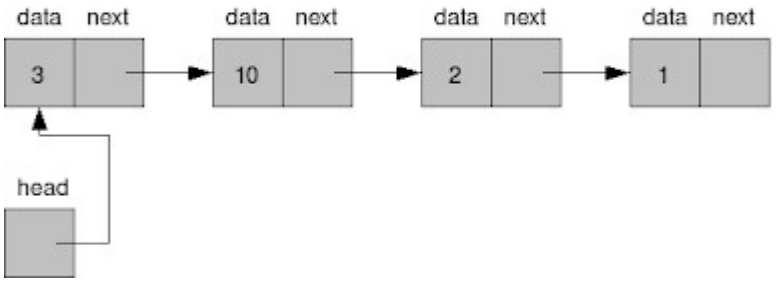
Esqueça a programação. Use seu bom senso e responda: o que é uma lista?
Vamos supor que você fez uma lista de compras.
Ou seja, tem um pedaço de papel com diversos elementos, em uma dada ordem.

Provavelmente essa ordem é a que você vai comprar, do primeiro para o último. Geralmente elas tem uma sequência lógica, como elementos do mesmo setores estarem adjacentes.
Depois você vai trabalhar com essa lista, seguir a ordem dela, marcar se comprou ou não, anotar o preço etc.

A analogia é parecida para uma lista em C.
Em programação, lista é uma série de elementos ligados. O primeiro é ligado no segundo, que é ligado no terceiro etc.

Iremos aprender como colocar elementos, tirar, mudar de posição. Como elas estão ligadas, basta que tenhamos o endereço (ponteiro) para o primeiro elemento da lista.

Ou seja, iremos estudar as chamadas listas encadeadas, que são itens 'alinhados' numa fila.



- **Filas em C (Queue)**

Outro importante conceito de estrutura de dados dinâmica são as filas, que são exatamente iguais as do mundo real.
Como funciona a fila de um caixa eletrônico?

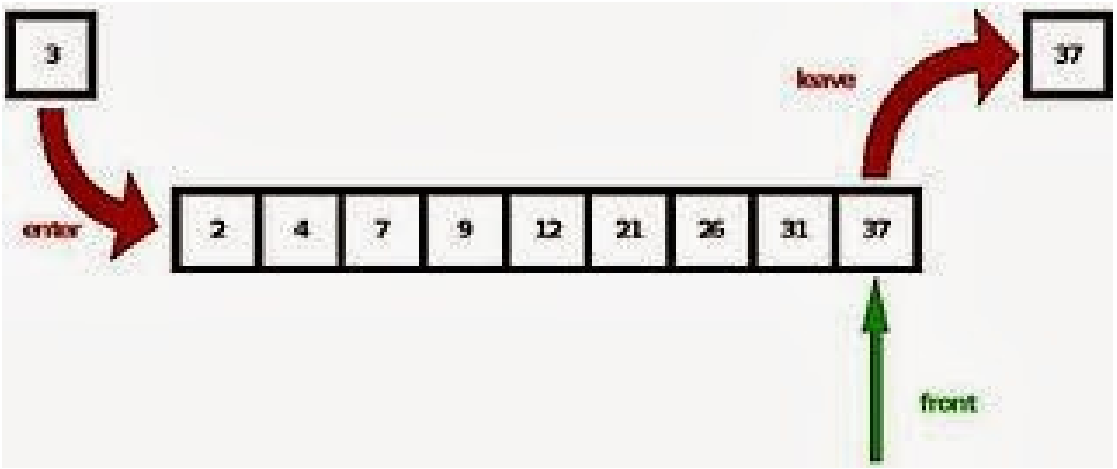
Chega a primeira pessoa, é atendida. Já a segunda, fica na fila, e será atendida depois da primeira.

A terceira a chegar só vai ser atendida depois da segunda, e assim sucessivamente.

Ou seja, os primeiros elementos a chegar serão os primeiros a serem atendidos.
Um termo muito conhecido para designar tal tipo de ideia é FIFO - First In, First Out (Primeiro que entra, primeiro que sai).

Em termos de programação, dizemos que os elementos que chegam vão para a cauda da fila, ou seja, para o final, serão atendidos por último. Os elementos que serão primeiro atendidos são os que estão na cabeça da fila (na frente).

Também chamamos o ato de colocar algo na fila de ENQUEUE, e de tirar de DEQUEUE.



- **Pilhas em C (Stack)**

Outra importante estrutura dinâmica de dados são as pilhas (stacks, em inglês), que tem um funcionamento contrário ao das filas. São ditas do tipo LIFO - Last In, First Out (Ultimo a entrar, primeiro a sair).

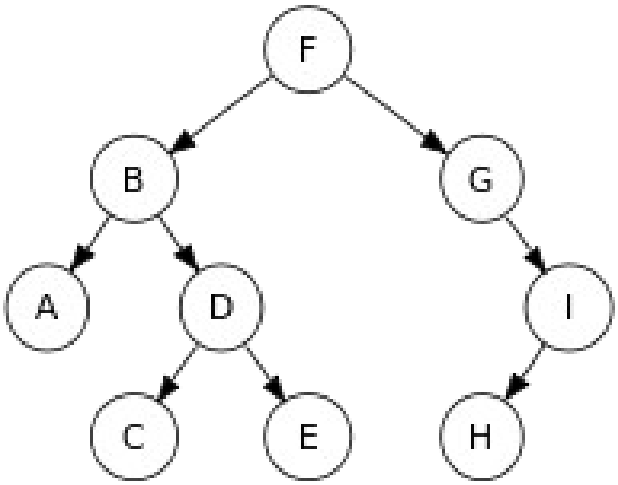
Para entender esse tipo de estrutura, podemos imaginar uma pilha de pratos. Você come uma coisa, e guarda o prato.

Come a segunda coisa, e põe o segundo prato em cima do primeiro.
Come a terceira e põe este prato em cima do segundo, e assim sucessivamente.

Quando você for lavar, que prato vai retirar primeiro? O de cima.
E qual o último prato a ser retirado? O primeiro.
Ou seja, você vai tirando os pratos de cima, que são os pratos que chegaram por último na pilha.
Já os pratos que chegaram primeiro na fila serão os últimos a saírem dela.

Dois termos comum nesse tipo de ordenação de estruturas são PUSH e POP.
PUSH é colocar um elemento na pilha, e POP é tirar.

- **Árvores em C (Tree)**



Os elementos anteriores geralmente apontam para um elemento (o próximo). Existem listas duplamente encadeadas, que cada elemento está ligado com o anterior e o próximo, mas todas elas são lineares, formam uma sequência única.

Já as árvores não, pois cada elemento pode estar ligado em um ou mais elemento, formando assim uma sub-árvore, e cada uma destas pode formar outras. Ou seja, há uma infinidade de caminhos e possibilidades para estruturas desse tipo.

Daremos atenção especial para as árvores binárias, que possuem no máximo duas ramificações.

- **Conceitos básicos de uma estrutura dinâmica de dados**

Antes de começarmos a estudar separadamente cada estrutura que foi introduzida neste tutorial, vamos ver um esqueleto, uma estrutura básica que iremos usar em todos os itens aqui abordados.

Primeiro de tudo, iremos realmente usar estruturas, ou seja, structs.
Isso porque são elementos bem gerais, que nos permitem criar qualquer tipo de dado, da maneira que quisermos.

A segunda ideia essencial para esse estudo, é a chamada auto-referência. Ou seja, vamos criar uma struct de um determinado tipo, e dentro dela iremos declarar um membro que é um ponteiro.

Como sabemos, os ponteiros apontam para tipos diferentes de dados, e nesse caso, o ponteiro que fica dentro de uma struct irá apontar para um tipo dessa mesma struct.

Isso se chama auto-referência. Pode ser um pouco estranho, mas é perfeitamente possível e uma ideia bastante usada.

Lembre-se: quando criamos uma struct, ela terá um tamanho (sizeof()) definido. Portanto, ela será um tipo de dado.

Então, é possível existir um ponteiro para este tipo de dado.
Em suma: é uma estrutura que possui um ponteiro.

Geralmente chamamos de nó (ou node, do inglês) os elementos de uma estrutura de dados dinâmica.
Veja como seria nossa struct:

```
struct node
{
 //dados e variáveis de sua struct aqui
 struct node *nextNode;
}
```

Esse ponteiro será de uso essencial, como veremos em breve em nossa apostila, pois eles irão apontar para o próximo elemento (próximo nó) ou para NULL, caso seja o último elemento da estrutura de dados.

É importante não se esquecer isso, é a terceira característica básica de todos esses tipos de estrutura de dados, pois é um erro bem comum entre iniciantes: o ponteiro sempre aponta para uma região de memória, aqui iremos fazer ele apontar para o próximo elemento da lista, pilha, fila ou árvore.

Quando criamos uma struct, seu ponteiro automaticamente vai apontar para o lixo, que é um endereço de memória como é endereço de memória caso apontemos esse ponteiro para outra struct, não há diferença.

Por isso é importante apontarmos o último elemento para NULL, pois diferente dos outros, só ele aponta para esse local, daí podemos usar essa informação para saber quando nossa estrutura de dados termina :)

Por hora, não se assuste. Iremos explicar, em detalhes, com muitos exemplos, cada um destes tipos, bem como faremos diversos algoritmos. Lembrando que é um assunto de suma importância, pois são ideias largamente usadas em programação.

Seu sistema operacional, por exemplo, usa e abusa de filas e pilhas para organizar os processos, o que deve ser executado primeiro, depois etc.

Listas em C - O que é e como funciona uma List

Estrutura de dados dinâmica realmente não é o assunto mais fácil da linguagem C, é realmente necessário estudar bastante para entender perfeitamente os conceitos e ideias por trás deste assunto.

Visando clarear mais a mente de nossos estudantes, a apostila C Progressivo vai explicar o conceito de lista apenas com figuras e explicações escritas, nada de código por hora.

- **Lista encadeada - O nó**

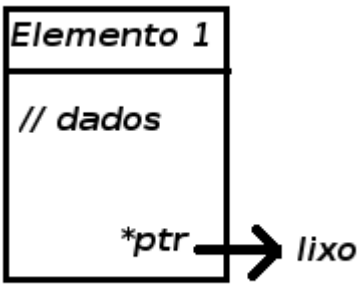
Vamos tentar explicar, sem pressa e com detalhes, cada uma das ideias e elementos necessários para se estudar e trabalhar com estruturas de dados dinâmicas.

O objetivo deste tutorial de nossa apostila é que você entenda, na sua cabeça, como tudo funciona.

E esta é uma dica valiosa: antes de programar e fazer seu aplicativo rodar, ele tem que rodar perfeitamente na sua cabeça. Ou seja: antes de começar, entenda e faça na mente.

Vamos chamar cada struct, a partir de agora, de **nó**.

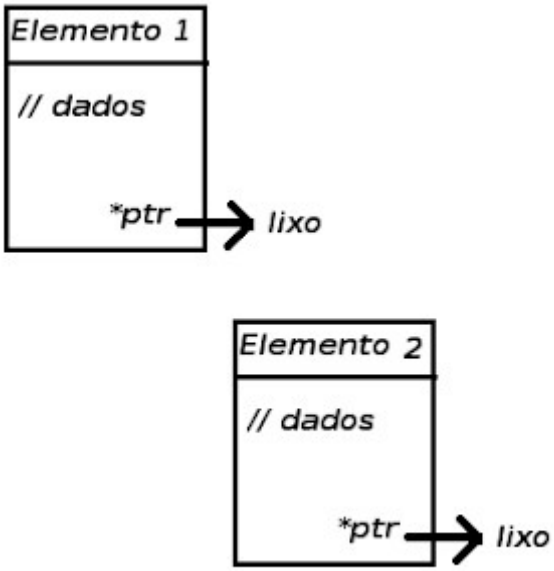
Por exemplo, criamos uma struct com algumas variáveis e um ponteiro que aponta para o seu próprio tipo. Logo, quando criamos tal nó, é da seguinte forma:



Pronto. Esse nó está em algum ponto da memória, e seu ponteiro aponta para um endereço de memória qualquer.

Agora vamos declarar outro nó, o *Elemento 2***.** Seu funcionamento e criação (alocando memória de maneira dinâmica) são idênticos ao *Elemento 1*, e essa estrutura, ou nó, está em um lugar qualquer da memória, bem como seu ponteiro aponta para um local aleatório, chamado de lixo.

Veja:



- **Conectando os nós de uma lista**

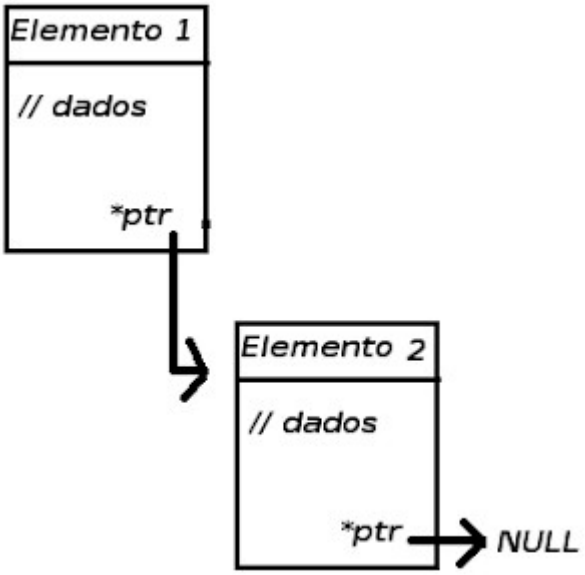
Agora temos que 'ligar', 'conectar' ou 'encadear' esses dois nós.

Lembre-se que o ponteiro é para o tipo do nó, então podemos fazer com que o ponteiro do primeiro nó aponte para o segundo nó! E como o nó foi alocado de maneira dinâmica, ele também é representado por um endereço de memória, que será usado pelo ponteiro do elemento 1.

No nó 1, fazemos:
ptr = Elemento2;

Porém, o ponteiro do segundo nó ainda aponta para outro endereço de memória (lixo).

Para podermos identificar que esse elemento é o último da lista, vamos fazer com que o ponteiro do nó 2 aponte para NULL, veja como ficou nossa lista encadeada:

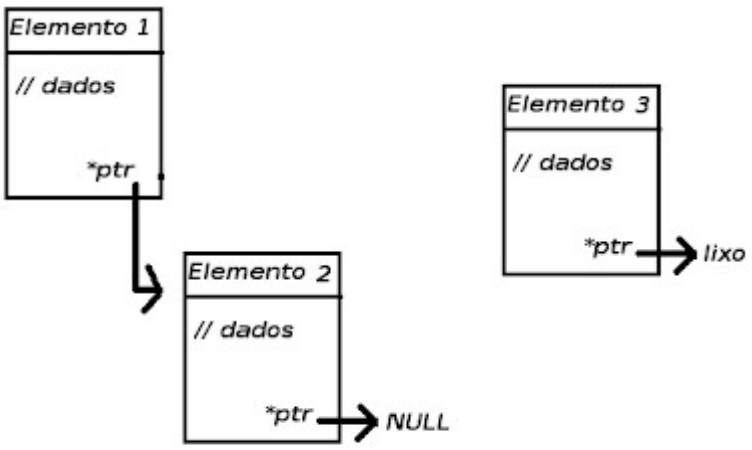


Pronto! Temos uma lista simples, com duas estruturas conectadas.

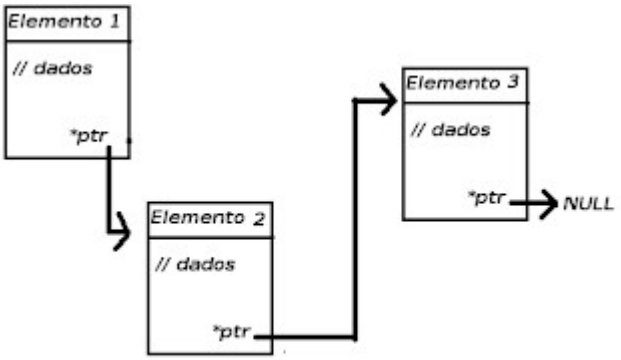
- **Inserindo nó ao final da lista**

Vamos colocar outro nó na lista?

Primeiro declaramos o elemento 3.



Como queremos que ele fique depois do elemento 2, fazemos com que o ponteiro deste segundo elemento aponte para o nó 3. E para identificar o nó 3 como o último da lista, fazemos seu ponteiro apontar para NULL. Veja:



Simples, não? Agora raciocine como seria para colocar um quarto nó ao final da lista. O que você faria?
Quem apontaria para quem? E qual ponteiro vai apontar para NULL agora?

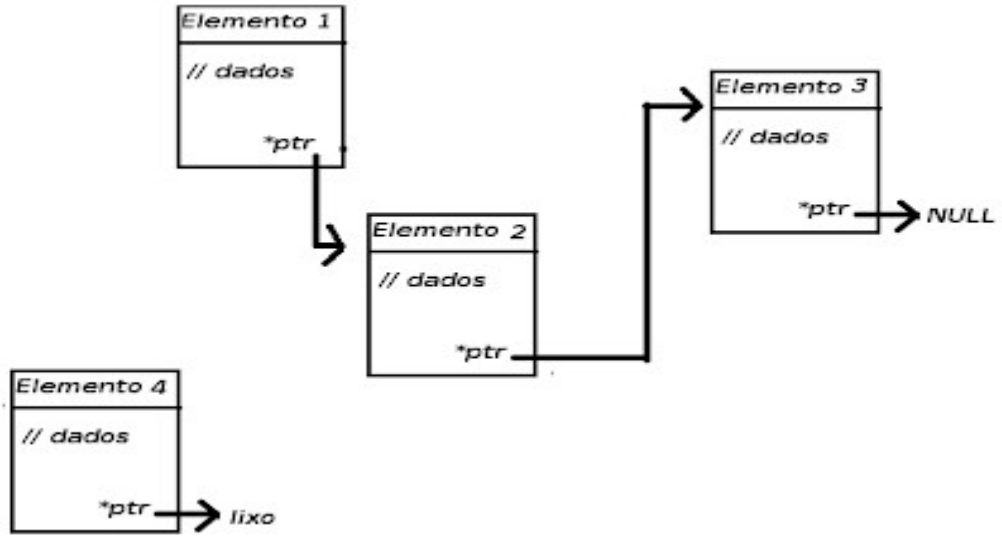
- **Conectando nós no início da lista**

Nos exemplos acima, conectamos nós ao final da lista.

Basicamente fazemos com que o nó antigo, que era o último, aponte para o novo nó, e o ponteiro desse novo nó aponte para NULL: é o nosso algoritmo para inserir um nó ao fim da lista.

Agora iremos aprender como inserir o nó no COMEÇO da lista, na primeira posição!

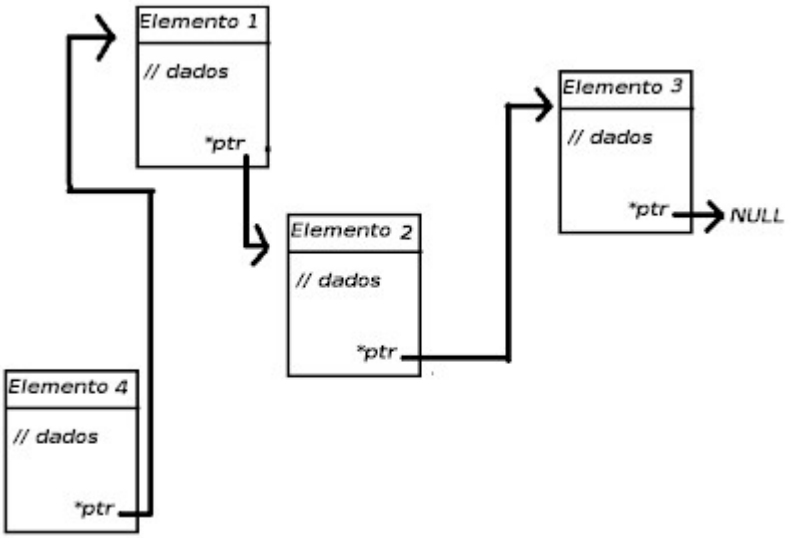
Como antes, primeiro declaramos um novo nó, o *Elemento 4*.
Ele está em algum lugar na memória, perdido e solitário, assim como o seu ponteiro.



Precisamos, agora, conectar esse novo nó na lista.

Porém, ninguém vai apontar para ele desta vez. Agora é ele que vai apontar para o primeiro nó da lista, que é sempre chamado de **head** (ou **cabeça**, em português).

Ou seja, no ponteiro do novo nó, fazemos:
ptr = Elemento1 ou ptr=cabeca



E pronto! Adicionamos o nó 4 no início da lista. Simples e óbvio, não?

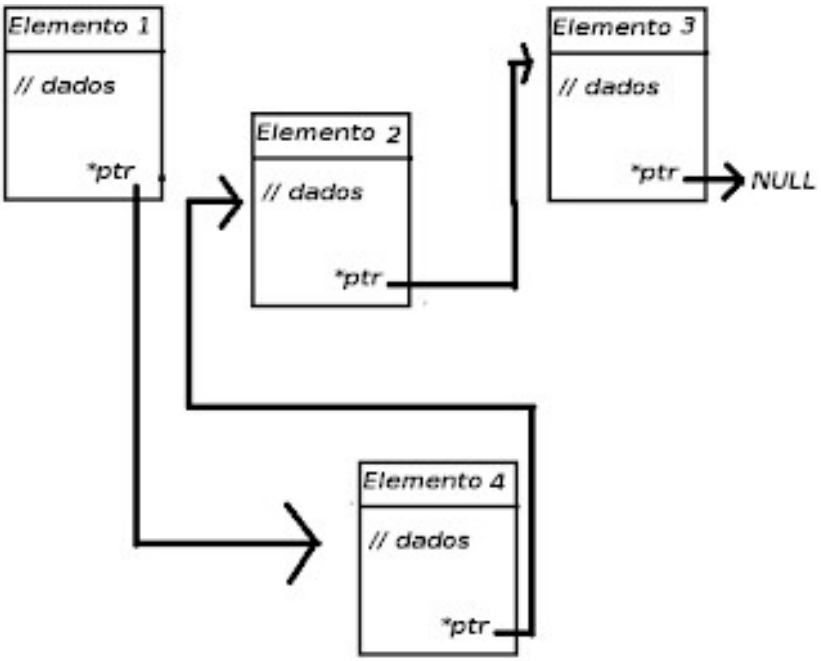
- **Inserindo um nó no meio da lista**

Vimos que para adicionar um nó no início da lista, basta o novo primeiro nó apontar para o antigo primeiro nó. Já para colocar ao fim da lista, simplesmente apontávamos o ponteiro do último nó para o novo nó.

E para adicionar no meio da lista? Fazemos as duas coisas.

Por exemplo, para colocar o nó na segunda posição: primeiro declaramos o nó 4, que está em um lugar qualquer da memória.
Agora fazemos com que o primeiro ponteiro deixe apontar para o nó 2 e aponte para o novo nó, o nó 4.

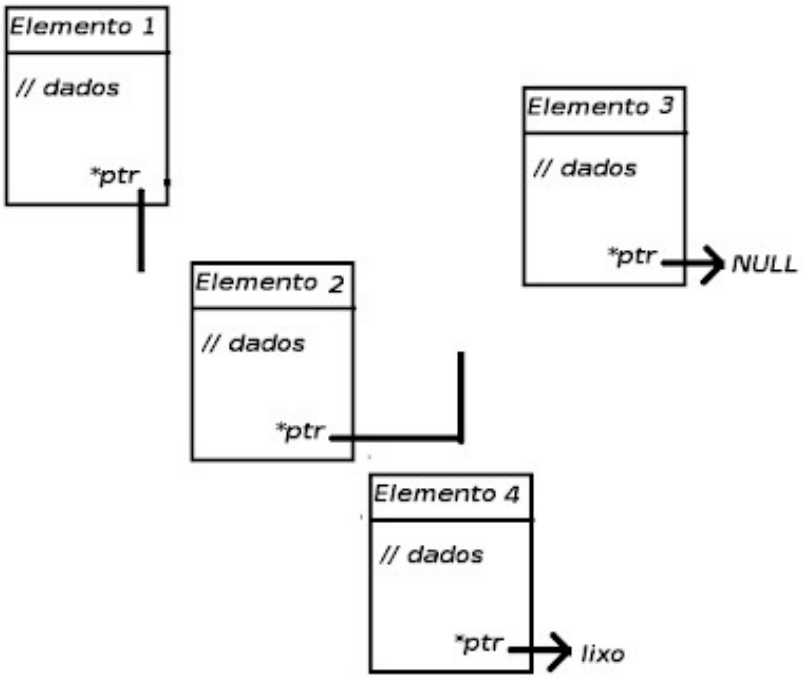
E agora fazemos com que o nó 4 aponte para o nó 2.



- **Excluindo um nó da lista**

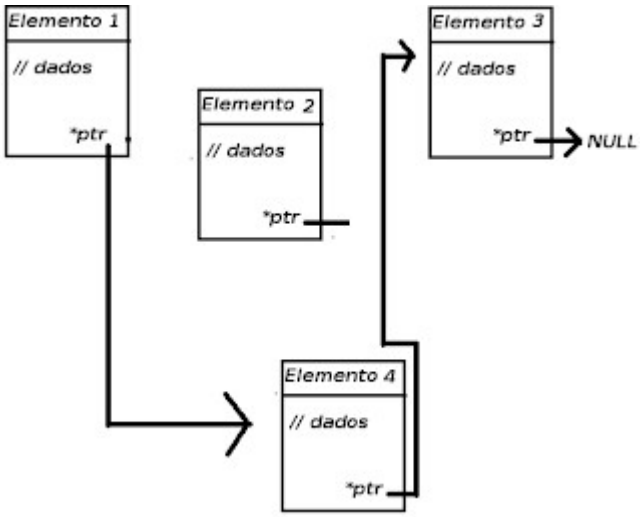
Para excluir um nó de uma lista, devemos fazer com que esse nó não aponte para ninguém e que ninguém da lista aponte para ele. Ou seja, ele não pode se relacionar com nenhum nó, deve se isolar.

No exemplo anterior, se quisermos tirar o nó 2, é só fazermos os passos contrários.



O nó 4 apontava para o 2. Vamos cortar essa relação: agora o nó 4 não vai mais apontar par ao nó 2, e sim para o nó 3. E ainda fazer com que o nó 2 que apontava para o nó 3, aponte para outro lugar, que não seja na lista.

Pronto. Ninguém aponta para o nó 2, e ele não aponta para ninguém.
O nó 1 aponta para o 4, que aponta para o 3, que aponta para NULL. Cadê o nó 2? Não faz mais parte, vai ficar lá perdido na memória, pois não vai se relacionar com ninguém da lista.



E para excluir o primeiro nó?
Ora, o nó 1 só faz apontar para o nó 2. Se fizermos ele apontar para NULL, ele sai da lista, já que ele é o primeiro e ninguém aponta para ele. Basta fazermos com que o nó 2 agora seja o **cabeça** e prontinho, bye bye nó 1.

E para excluir o último nó ?
Basta fazer com que o penúltimo nó, que apontava para o último, deixe de apontar para ele e aponte para NULL. Pronto, agora ninguém aponta pro último e ele ficou apontando para NULL, logo, não tem mais nenhuma relação com a lista encadeada.

Simples, bonitinho, com desenhos e tudo fazendo sentido.
Se não entender, leia, releia até entender. Só podemos ir para o próximo passo depois de entendido perfeitamente este tutorial de nosso curso.

Caso tenha entendido, é hora de partir para os códigos em C.
No próximo tutorial vamos ensinar como programar uma lista encadeada em C. Até lá.

Lista simplesmente encadeada com cabeça em C: Inserindo nós no início e no fim

Agora que já vimos como funciona uma lista em C, vamos aprender como implementar, como programar uma lista desde o início.

Como há diversas maneiras de se fazer isso, pois há diversos tipos de listas, vamos fazer vários tipos de listas e, aos poucos, iremos mostrar as ideias que podemos utilizar para criar mais recursos para nossas listas.

- **Tipos de Listas em C**

No esquema que mostramos no tutorial passado de nossa apostila de C, explicamos como funciona uma lista, como seria adicionar um elemento (nó) em seu início, no seu fim ou em qualquer posição da lista.

Também vimos como retirar um elemento da lista, tanto do início, do fim como de uma posição intermediária.

Só aí já iriam várias implementações destas ideias.

E além destes recursos, há os vários tipos de listas.
Vamos estudar as simplesmente encadeadas, ou seja, seus nós são structs que possuem só um ponteiro, que aponta sempre para o elemento seguinte da lista.

Além deste tipo, que é o mais simples, há ainda as listas duplamente encadeadas, que possuem ponteiros para o próximo e para o nó anterior. Há também as listas circulares, onde o último elemento se liga ao primeiro, e por aí vai.

- **Lista encadeada com cabeça**

Neste artigo iremos focar nas simplesmente encadeadas, e sobre como inserir nós no início e ao fim da lista.

Embora sejam 'poucas' coisas, você verá que na verdade, na hora de implementar pela primeira vez, pode parecer algo bem difícil e confuso.

Mas iremos progredir aos poucos. Depois iremos aprender como excluir os elementos do fim, do início, bem como inserir ou retirar de qualquer local da lista, e até mesmo buscar nós específicos.

Mas sem pressa, por hora, vamos fazer o básico, que só isso já vai dar mais de 150 linhas de código.

Podemos implementar uma lista simplesmente encada de duas maneiras: com cabeça e sem cabeça.
Onde a cabeça de uma lista seria um nó, que foi declarado explicitamente no início do programa.

Não usaremos, na verdade, o conteúdo deste nó, também chamado de célula ou cabeça, pois como veremos, ele não será relevante.
Ele serve para sinalizar o início da lista, para sabermos que ela inicia ali, naquele endereço fixo de memória.

Fazemos isso na primeira linha de nossa main():
node *LISTA = (node *) malloc(sizeof(node));

Como explicamos no tutorial passado, esse 'node' é um tipo que definimos (typedef struct), uma estrutura que armazena um número inteiro e um ponteiro para o tipo 'node' (nó, em inglês).

A lista, de fato, será colocada no ponteiro deste nó inicial (cabeça), que está em: LISTA→prox

Como acabamos de criar a lista, esse ponteiro deverá apontar para NULL, para sinalizar que a lista está vazia.
Vamos fazer isso através do método 'inicia()', que simplesmente faz: LISTA->prox = NULL;

Criamos um método para isto, pois este simples ato faz nossa lista ser zerada, sendo possível ser preenchida de novo.

Em seguida, invocamos o método menu(), que irá mostrar ao usuário uma série de opções e irá armazenar a opção escolhida, que será enviada ao método 'opcao()', que irá tratar a opção escolhida através de um teste condicional SWITCH.

- **Como inserir um nó no início da lista**

Inserir nós no início da lista é bem simples de se fazer, pois temos um head node, ou seja, um nó cabeça, que nos indica onde está o início de nossa lista: LISTA->prox

Primeiro precisamos criar o nosso novo nó, o 'novo'.

Agora devemos fazer dois passos: primeiros fazemos LISTA->prox apontar para nosso novo nó, depois fazemos nosso nó apontar para o próximo elemento da lista.

Mas aqui temos um problema.
Antes o 'LISTA->prox' apontava para um local de memória que tinha o próximo nó.
Mas quando apontamos ele para o nó 'novo', perdemos essa referência.

Vamos resolver isso criando um ponteiro para armazenar o local que está 'LISTA->prox' antes de inserirmos o novo nó.
Vamos chamar de 'oldHead', pois era a cabeça antiga da lista:
node *oldHead = LISTA->prox;

Então, agora apontamos a cabeça da lista para nosso novo nó: LISTA->prox = novo;
E apontamos nosso novo nó para o nó antigo: novo->prox = oldHead;

E pronto, inserimos uma estrutura no início da lista. Interessante, não?

- **Como inserir um nó ao final da lista**

Vamos agora inserir um nó ao final da lista.

Antes de mais nada, devemos alocar memória para este nó e preencher o número que iremos armazenar nele, o que é facilmente feito através da função malloc(), conforme estudamos na seção de nosso curso de Alocação Dinâmica de Memória:

node *novo=(node *) malloc(sizeof(node));

Lembrando sempre de checar se a memória foi alocada (caso não, devemos dar um exit() no programa, informando o erro).

Agora devemos procurar o final da lista.

Antes, devemos checar se ela está vazia, através do método vazia(), que simplesmente checa o elemento: LISTA→prox

Se este apontar para NULL, a lista está vazia, afinal, esse elemento é a cabeça da lista, aponta par ao início dela.

Caso seja realmente vazia, vamos fazer com que a cabeça aponte para este novo nó que criamos, o 'novo':
LISTA->prox = novo;

Agora, este nó será o último elemento da lista, então ele DEVE apontar para NULL:
novo->prox = NULL;

E caso não seja uma lista vazia? Bem, devemos procurar o final da lista, que é um nó que aponta para NULL.

Para isso, vamos declarar um ponteiro de nó:
node *tmp;

A ideia é fazer esse ponteiro apontar para todos os elementos da lista e checar se ele é o último.

Algo como "Hey, você é o último da lista? Não? Próximo! E você, é? Não? Ok, próximo..."

Então posicionamos ele apontando para o início da lista, nossa cabeça: tmp = LISTA->prox
Agora vamos percorrer todos os nós, e só vamos parar quando o nó apontar para NULL, ou seja, só vamos parar quando a seguinte comparação retornar valor lógico TRUE:

tmp->prox == NULL

Se retornar falso, devemos avançar a lista, e isso é feito da seguinte maneira:
tmp = tmp->prox;

Isso é feito com um laço while, que sai varrendo todos os nós e só para quando encontra o último:
```
while(tmp->prox != NULL)
   tmp = tmp->prox;
```

Ou seja, nosso ponteiro agora aponta para o próximo elemento da lista.
E quando achar o ponteiro que aponta para NULL? Aí colocamos aquele nó que criamos, o 'novo'.
Ou seja, em vez do último nó apontar para NULL, vai apontar para novo:
tmp->prox = novo;

E como, ao declarar o 'novo', fizemos ele apontar para NULL, identificamos ele como o último elemento da lista.
Genial, não?

- **Como exibir os elementos de uma lista**

Bom, em nada adianta alocar os elementos no início ou fim da lista, se não pudermos ver essa lista crescendo diante de nossos olhos.
Por isso vamos criar uma função que será responsável por exibir todos os elementos da lista, do início ao fim.

E onde é o início da lista? Ué, no nó cabeça: LISTA->prox
E onde é o final? É em um nó que aponta para NULL.

Mais uma vez precisamos criar um ponteiro que vai apontar para cada um dos elementos da lista, e depois imprimir o número armazenado na estrutura.

Vamos chamar, novamente, esse nó temporário de 'tmp' e deve apontar, inicialmente para a cabeça da lista:
node *tmp = LISTA->prox;

Agora vamos imprimir o primeiro elemento (caso exista, senão existir, dizemos que a lista está vazia).
Estamos em um nó, então imprimimos ele e avançamos.

Para imprimir, damos um print no inteiro: tmp->num
E para avançar na lista: tmp = tmp->prox

E quando devemos parar o avanço? Quando chegarmos ao fim da lista. E quando isso ocorre? Quando nosso ponteiro temporário apontar para NULL. Logo, isso é feito dentro de um laço while:
while(tmp != NULL)

Pronto, membros da lista exibidos.

- **Liberando a memória armazenada para uma lista**

Como enfatizamos bem ao ensinar o uso da função free() para liberar a memória, sempre devemos devolver o que foi alocado para nosso sistema, sob risco de ocorrer vazamento de memória que podem atrapalhar e travar um computador.

Na lista, as coisas são um pouco mais complexas, pois alocamos memória várias vezes!
Portanto, devemos sair desalocando e liberando cada nó que alocamos!

Bom, vamos lá, mais uma vez usar nosso ponteiro para nó (caso não seja uma lista vazia), apontar para cada estrutura da lista e dar um free() nela. Vamos chamar esse ponteiro para nó de 'atual'.

Inicialmente aponta para a cabeça: node *atual = LISTA->prox;
Agora libera a memória: free(atual);
Indo para o próximo elemento: atual = atual->prox;
Ok?

Não, isso está errado. Lembre-se que se liberamos o primeiro nó e junto com ele o seu ponteiro *prox.
Ou seja, não temos mais o local do próximo nó. E agora, José?

A solução é, antes de liberar um nó, salvar seu ponteiro que está apontando para o próximo nó.
Vamos criar um ponteiro para nó para guardar essa informação, será o 'proxNode'.

Vamos começar de novo, agora salvando o próximo nó antes de desalocá-lo.
Recebe o primeiro nó: atual = LISTA->prox;
Guarda o endereço do próximo nó: proxNode = atual->prox;
Desaloca: free(atual);

Pronto. Desalocamos o nó, e sabemos onde está o próximo (guardado no ponteiro 'proxNode').
Agora esse nó será o próximo a ser desalocado, basta repetir o procedimento:
Recebe o próximo nó: atual = proxNode;
Guarda o endereço do próximo nó: proxNode = atual->prox;
Desaloca: free(atual);

E assim sucessivamente. E quando isso deve parar? Quando chegar ao final.
E onde é o final da lista? É no nó que aponta para NULL.

Ou seja, vamos avançando na lista, e quando estivermos em um nó que é NULL, paramos, não tentamos desalocá-lo.
Assim, só liberamos memória quando o nó atual não for NULL: while(atual != NULL), e dentro desse while vão aqueles procedimentos que falamos (guarda o endereço do próximo, desaloca e vai para o próximo).

- **Código em C de uma lista encadeada**

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

struct Node{
	int num;
	struct Node *prox;
};
typedef struct Node node;

void inicia(node *LISTA);
int menu(void);
void opcao(node *LISTA, int op);
node *criaNo();
void insereFim(node *LISTA);
void insereInicio(node *LISTA);
```

```c
void exibe(node *LISTA);
void libera(node *LISTA);


int main(void)
{
	node *LISTA = (node *) malloc(sizeof(node));
	if(!LISTA){
		printf("Sem memoria disponivel!\n");
		exit(1);
	}
	inicia(LISTA);
	int opt;

	do{
		opt=menu();
		opcao(LISTA,opt);
	}while(opt);

	free(LISTA);
	return 0;
}

void inicia(node *LISTA)
{
	LISTA->prox = NULL;
}

int menu(void)
{
	int opt;

	printf("Escolha a opcao\n");
	printf("0. Sair\n");
	printf("1. Exibir lista\n");
	printf("2. Adicionar node no inicio\n");
	printf("3. Adicionar node no final\n");
	printf("4. Zerar lista\n");
	printf("Opcao: "); scanf("%d", &opt);

	return opt;
}

void opcao(node *LISTA, int op)
{
```

```
        switch(op){
                case 0:
                        libera(LISTA);
                        break;

                case 1:
                        exibe(LISTA);
                        break;

                case 2:
                        insereInicio(LISTA);
                        break;

                case 3:
                        insereFim(LISTA);
                        break;

                case 4:
                        inicia(LISTA);
                        break;

                default:
                        printf("Comando invalido\n\n");
        }
}

int vazia(node *LISTA)
{
        if(LISTA->prox == NULL)
                return 1;
        else
                return 0;
}


void insereFim(node *LISTA)
{
        node *novo=(node *) malloc(sizeof(node));
        if(!novo){
                printf("Sem memoria disponivel!\n");
                exit(1);
        }
        printf("Novo elemento: "); scanf("%d", &novo->num);
        novo->prox = NULL;
```

```
        if(vazia(LISTA))
                LISTA->prox=novo;
        else{
                node *tmp = LISTA->prox;

                while(tmp->prox != NULL)
                        tmp = tmp->prox;

                tmp->prox = novo;
        }
}

void insereInicio(node *LISTA)
{
        node *novo=(node *) malloc(sizeof(node));
        if(!novo){
                printf("Sem memoria disponivel!\n");
                exit(1);
        }
        printf("Novo elemento: "); scanf("%d", &novo->num);

        node *oldHead = LISTA->prox;

        LISTA->prox = novo;
        novo->prox = oldHead;
}

void exibe(node *LISTA)
{
        if(vazia(LISTA)){
                printf("Lista vazia!\n\n");
                return ;
        }

        node *tmp;
        tmp = LISTA->prox;

        while( tmp != NULL){
                printf("%5d", tmp->num);
                tmp = tmp->prox;
        }
        printf("\n\n");
}

void libera(node *LISTA)
```

```
{
	if(!vazia(LISTA)){
		node *proxNode,
		     *atual;

		atual = LISTA->prox;
		while(atual != NULL){
			proxNode = atual->prox;
			free(atual);
			atual = proxNode;
		}
	}
}
```

- **Considerações sobre listas em C**

Sim, não é fácil.
Ponteiros não são fáceis, alocação também não é, e obviamente se juntarmos os dois, vai dar algo que realmente não é simples. Porém, não é impossível.

É perfeitamente natural não entender de primeira.
Leia, releia, veja mais uma vez. Clareie a mente, depois volte a estudar de novo.
Tente realmente entender cada passo, tente refazer. Pesquise na internet, leia o livro que indicamos.
Vai levar um tempo até se habituar com essas ideias, mas acredite, acontece com todos.

Essas ideias são EXTREMAMENTE poderosas!
Note que estamos tratando de estruturas, e essas structs podem ser qualquer coisa.

Podem guardar dados de funcionários de uma empresa, informações de aluno de uma escola ou cada uma pode ser a bula de um remédio de uma farmácia, no sistema.

Se tivermos 10 funcionários, as listas alocam espaço somente para estes 10. Se forem 100 alunos, teremos espaço para exatos 100 alunos. E se a farmácia tiver mil tipos de remédios, só vamos usar mil espaços de struct. Nem mais, nem menos.

Precisa guardar mais uma informação? Somente mais uma struct será usada.

E como veremos no próximo tutorial, quando não precisarmos de mais um elemento da lista, liberaremos seu espaço de memória.

Resumindo: é um algoritmo extremamente eficiente, consome o mínimo de memória possível, e acredite: esta ideia está sendo usada agora, em seu sistema operacional.

Não desanime, continue estudando.

Exercícios sobre listas

No próximo tutorial sobre listas de nossa apostila de C, iremos ensinar como retirar um elemento do início, um elemento do fim e um intermediário.
Tente implementar estes recursos.

Lista simplesmente encadeada: Retirando nós no início e no fim

Como já aprendemos a inserir elementos no início e no fim de uma lista, nada mais justo que aprender como retirar nós do início e do fim de uma lista., que é o que iremos aprender neste tutorial de nossa apostila C Progressivo.

No próximo tutorial faremos uma generalização, mostrando como inserir e retirar elementos de qualquer posição da lista, e teremos uma lista completa e flexível.

- **Retirando nós da uma lista em C**

Agora vamos aprender como retirar estruturas de uma lista.
Assim como fizemos para inserir, vamos aprender como retirar no início e no fim, que são mais simples e ajudará você a entender melhor.

Depois, quando entender melhor esses conceitos, podemos fazer algo mais completo, que é retirar e inserir uma struct em qualquer posição.

Uma diferença importante é que, quando inseríamos nós, nossas funções eram do tipo ***void***, afinal, simplesmente tínhamos que colocar um nó ali na lista.

Mas quando retiramos, é importante saber que elementos retiramos.
A explicação disso é o bom senso: quando tiramos um elemento de uma lista, é para trabalhar em cima dele, ver seu conteúdo, passar ele para outra função ou extrair algum dado.

Portanto, nossa funções de retirar elementos irão retornar o nó, serão do tipo:

```
node *retiraInicio(node *LISTA);
node *retiraFim(node *LISTA);
```

- **Retirando um nó do início da lista**

A primeira coisa que devemos fazer para retirar um nó do início é saber se esse nó realmente existe, ou seja, se a lista não está vazia.

Essa simples verificação é feita usando um teste condicional IF: if(LISTA->prox == NULL)

E caso dê verdadeira, é porque ela está vazia e não há para ser retirado.
Como devemos retornar algo, iremos fazer: return NULL;
Isso é necessário para mostrar que nada foi retirado da lista.

Agora iremos fazer a retirada do primeiro elemento da lista através do uso de um ponteiro para o tipo ***node***, o *tmp
Vamos fazer ele apontar para primeiro elemento da lista: tmp = LISTA->prox;

Então o primeiro elemento da lista está apontado por 'tmp'.
Agora vem a lógica da exclusão de um elemento.

Antes, LISTA->prox aponta para o primeiro elemento, então para excluir esse primeiro elemento, basta fazer com que esse ponteiro aponte para o segundo elemento.
E onde está o segundo elemento? Está em: tmp->prox;

Logo, para excluir o primeiro nó, fazemos: LISTA->prox = tmp->prox
E em seguida retornamos 'tmp', que ainda aponta para aquele nó que foi excluído.
Vale ressaltar que, embora tenhamos tirado ele da lista (fazendo com que ninguém aponte para ele), memória foi alocada para esse nó, então ele ainda existe, e está sendo apontado por 'tmp'.

- **Como excluir um elemento ao final da lista**

Assim como fizemos para tirar uma struct do início, para tirar do fim primeiro temos que checar se a lista está vazia, que é análogo à maneira anteriormente mostrada.

Se notar bem, para retirar um elemento do início da lista, usamos dois ponteiros: LISTA->prox e tmp

Para fazer isso no final da lista, também vamos usar dois ponteiros.

Porém, não vamos poder usar o LISTA->prox pois este está apontado para o início da lista.

Para resolver este problema, iremos usar dois ponteiros neste algoritmo: 'ultimo' e 'penultimo'

Como os próprios nomes podem sugerir, o 'ultimo' aponta para último elemento da lista e o 'penultimo' para o penúltimo elemento da lista.

Nosso objetivo é simples, fazer o ponteiro 'ultimo' chegar ao fim da lista e o 'penultimo' a penúltima posição. Chegando lá, simplesmente fazemos o ponteiro 'penultimo' apontar para NULL, caracterizando este elemento como o último da lista, e retornamos o ponteiro 'ultimo', que contém o elemento retirado.

E como sabemos que um elemento é o último da lista?
Simples, ele aponta para NULL. Então vamos fazer o ponteiro 'ultimo' ir percorrendo a lista até chegar num ponto onde 'ultimo->prox' aponta para NULL, e aí chegamos ao fim.

Porém, antes do 'ultimo' avançar, devemos fazer com que o ponteiro 'penultimo' receba o valor de 'ultimo'. Colocando isso dentro de um laço while, obtemos o que queríamos da seguinte maneira:

```
while(ultimo->prox != NULL){
	penultimo = ultimo;
	ultimo = ultimo->prox;
}
```

Agora vamos excluir o último da lista, fazendo o penúltimo elemento apontar para NULL, e retornar ele:

```
penultimo->prox = NULL;
return ultimo;
```

- **Código C da uma Lista encadeada**

Logo, o código de um programa em C que implementa uma lista, onde é possível adicionar ou retirar elementos tanto do início como do fim, é:

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

struct Node{
	int num;
	struct Node *prox;
};
typedef struct Node node;


void inicia(node *LISTA);
int menu(void);
void opcao(node *LISTA, int op);
node *criaNo();
void insereFim(node *LISTA);
void insereInicio(node *LISTA);
void exibe(node *LISTA);
void libera(node *LISTA);
node *retiraInicio(node *LISTA);
node *retiraFim(node *LISTA);


int main(void)
{
	node *LISTA = (node *) malloc(sizeof(node));
	if(!LISTA){
		printf("Sem memoria disponivel!\n");
		exit(1);
	}else{
	inicia(LISTA);
	int opt;

	do{
		opt=menu();
		opcao(LISTA,opt);
	}while(opt);

	free(LISTA);
	return 0;
	}
}
```

```c
void inicia(node *LISTA)
{
	LISTA->prox = NULL;
}

int menu(void)
{
	int opt;

	printf("Escolha a opcao\n");
	printf("0. Sair\n");
	printf("1. Zerar lista\n");
	printf("2. Exibir lista\n");
	printf("3. Adicionar node no inicio\n");
	printf("4. Adicionar node no final\n");
	printf("5. Retirar do inicio\n");
	printf("6. Retirar do fim\n");
	printf("Opcao: "); scanf("%d", &opt);

	return opt;
}

void opcao(node *LISTA, int op)
{
	node *tmp;
	switch(op){
		case 0:
			libera(LISTA);
			break;

		case 1:
			libera(LISTA);
			inicia(LISTA);
			break;

		case 2:
			exibe(LISTA);
			break;

		case 3:
			insereInicio(LISTA);
			break;

		case 4:
			insereFim(LISTA);
```

```
                break;

            case 5:
                tmp= retiraInicio(LISTA);
                printf("Retirado: %3d\n\n", tmp->num);
                break;

            case 6:
                tmp= retiraFim(LISTA);
                printf("Retirado: %3d\n\n", tmp->num);
                break;

            default:
                printf("Comando invalido\n\n");
        }
}

int vazia(node *LISTA)
{
        if(LISTA->prox == NULL)
                return 1;
        else
                return 0;
}

node *aloca()
{
        node *novo=(node *) malloc(sizeof(node));
        if(!novo){
                printf("Sem memoria disponivel!\n");
                exit(1);
        }else{
                printf("Novo elemento: "); scanf("%d", &novo->num);
                return novo;
        }
}


void insereFim(node *LISTA)
{
        node *novo=aloca();
        novo->prox = NULL;

        if(vazia(LISTA))
                LISTA->prox=novo;
```

```
        else{
                node *tmp = LISTA->prox;

                while(tmp->prox != NULL)
                        tmp = tmp->prox;

                tmp->prox = novo;
        }
}

void insereInicio(node *LISTA)
{
        node *novo=aloca();
        node *oldHead = LISTA->prox;

        LISTA->prox = novo;
        novo->prox = oldHead;
}

void exibe(node *LISTA)
{
        if(vazia(LISTA)){
                printf("Lista vazia!\n\n");
                return ;
        }

        node *tmp;
        tmp = LISTA->prox;

        while( tmp != NULL){
                printf("%5d", tmp->num);
                tmp = tmp->prox;
        }
        printf("\n\n");
}


void libera(node *LISTA)
{
        if(!vazia(LISTA)){
                node *proxNode,
                        *atual;

                atual = LISTA->prox;
                while(atual != NULL){
```

```
                proxNode = atual->prox;
                free(atual);
                atual = proxNode;
            }
        }
}


node *retiraInicio(node *LISTA)
{
        if(LISTA->prox == NULL){
            printf("Lista ja esta vazia\n");
            return NULL;
        }else{
            node *tmp = LISTA->prox;
            LISTA->prox = tmp->prox;
            return tmp;
        }

}

node *retiraFim(node *LISTA)
{
        if(LISTA->prox == NULL){
            printf("Lista ja vazia\n\n");
            return NULL;
        }else{
            node *ultimo = LISTA->prox,
                *penultimo = LISTA;

                while(ultimo->prox != NULL){
                    penultimo = ultimo;
                    ultimo = ultimo->prox;
                }

                penultimo->prox = NULL;
                return ultimo;
        }
}
```

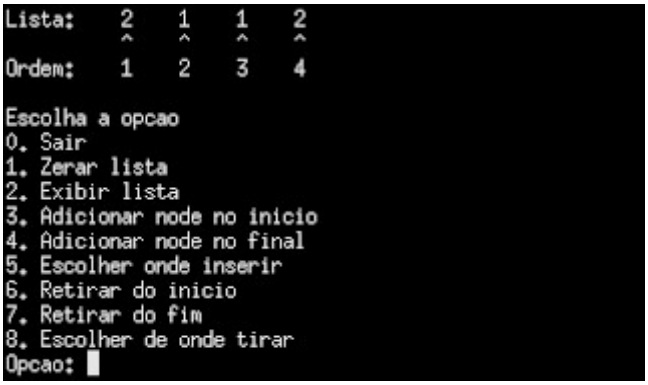
- **Exercício de C**

No próximo tutorial de nosso curso de C, vamos ensinar como inserir e retirar elementos de qualquer posição de uma lista, finalizando complemente nossa implementação de uma lista em C.

Tente fazer essas implementações, que serão explicadas no próximo tutorial de nosso curso de C.

Lista simplesmente encadeada: Inserindo e retirando nós de qualquer posição

Dando continuidade em nossa seção sobre estrutura dinâmica de dados e ao tutorial passado sobre Listas simplesmente encadeadas, onde criamos e ensinamos a colocar nós ao fim e no início da lista, e depois como retirar nós do início e do fim de uma lista, vamos agora mostrar como colocar elementos em qualquer ponto da lista, bem como tirar nós do início, do fim e de qualquer lugar da lista.



- **Como criar uma lista completa em C**

Este é o quarto artigo de nossa série sobre listas simplesmente encadeadas com cabeça.

Aqui mostraremos como inserir e retirar qualquer nó de nossa lista, e com isso temos um código completo, totalmente explicado, passo-a-passo sobre como criar esse tipo de estrutura de dados em C.

É o mais completo, com todo o código. Porém é necessário que você tenha lido os outros tutoriais de nossa apostila de C para entender melhor o que será explicado aqui.

Alterações em nossa lista

Agora que você já deve ter se familiarizado com os conceitos e lógicas de tirar e inserir elementos em uma lista, vamos fazer algumas mudanças, deixar o programa mais robusto e flexível.

Já no tutorial passado, na parte de inserir elementos, criamosa a função *aloca()*, que como o próprio nome diz, ela vai alocar espaço para uma estrutura, um nó em nossa lista.

Ela pede para inserir o número, colocar na variável 'num' da struct e retorna o endereço alocado dinamicamente.

Agora, criamos também a variável global ***tam***, que irá armazenar o tamanho de nossa lista.

Ou seja, quantos nós tem nossa lista. E para contabilizar isso, toda vez que criamos um nó, incrementamos essa variável.
Analogamente, sempre que tiramos uma struct da lista, decrementamos a variável.

Alteramos também a função exibe, que agora exibe a ordem dos elementos na lista.
Conforme podemos ver na imagem no início deste tutorial.

- ## Como inserir nós em qualquer posição da lista

Vamos agora mostrar como criar a função *insere()* que recebe a LISTA e pergunta ao usuário em que posição o usuário quer inserir o elemento na lista. Ou seja, se queremos inserir na posição 'n', o elemento vai ficar nessa posição 'n' e o que estava lá antigamente vai para frente, para posição 'n+1'.
O usuário vai dizer a posição e está sera armazenada na variável **pos**.

Podemos inserir desde a posição 1 até a 'tam'.

Obviamente, fora desse intervalo devemos mostrar uma mensagem de erro.
Feita essa verificação da posição, vamos adicionar o elemento na dita posição.

Caso seja posição 1, não devemos nos estressar.
Afinal, inserir um elemento na posição 1 é inserir uma estrutura no início da lista, e já criamos uma função para isto, a ***insereInicio()***, bastando chamar ela: insereInicio(LISTA);

Caso seja em qualquer outra posição, a coisa fica mais trabalhosa.
O segredo para isto é identificar dois elementos.o anterior e o elemento que está naquela posição.

Por exemplo, se queremos colocar um nó na terceira posição, devemos guardar essa posição e a anterior, pois iremos fazer o segundo elemento apontar para o novo nó, e fazer esse novo nó apontar para aquele que estava na terceira posição.

Para isso vamos usar dois ponteiros do tipo ***node***, o tipo de nossa estrutura: o 'atual' e o 'anterior'.
O atual começa no primeiro nó da LISTA, e o anterior não está em uma posição anterior (um aponta para LISTA->prox e o outro para LISTA).

Agora temos que fazer estes dois pontos 'correrem' pela lista até chegar onde queremos.

Vamos usar um laço for para isso, e em cada iteração fazemos o seguinte: Fazemos o ponteiro 'anterior' receber o ponteiro 'atual', e depois fazemos o 'atual' receber o próximo elemento da lista, que é o 'atual->prox'.

Se queremos chegar na posição 4, por exemplo, devemos fazer esse procedimento 3 vezes, pois partimos da primeira posição da lista. Ou seja, fazemos isso **(pos - 1)** vezes, e ao final deste procedimento, o ponteiro 'atual' estará no elemento que mudará de posição (para frente), e o ponteiro 'anterior' apontará para a posição anterior:

```
for(count=1 ; count < pos ; count++){
        anterior=atual;
        atual=atual->prox;
}
```

E agora vamos inserir nosso nó, que criamos ao declarar o ponteiro 'novo' e fazer ele receber um bloco alocado pela função aloca().
Vamos lá, devagar pois não é tão simples.

Temos dois nós: o 'atual' e o 'anterior'. Queremos colocar um novo nó, o 'novo', no lugar do nó 'atual' e empurrar o 'atual' pra frente.
Para isso, devemos pegar o 'anterior' e fazer apontar para o 'novo': anterior->prox = novo;
E o novo nó deve apontar para o que estava nesse posição: novo->prox = atual;
E claro, incrementar o tamanho da lista: tam++;

Pronto. Já podemos colocar um nó no início, no fim ou em qualquer lugar de nossa lista :)
Vamos para o próximo passo: retirar elementos de nossa lista.

- **Como retirar estruturas de uma lista**

Se já leu todos nossos tutoriais sobre listas em C, certamente já deve ter em mente como implementar um algoritmo que retire um elemento.

Vamos usar exatamente a mesma ideia que usamos na parte passada do tutorial, para achar os elementos 'atual' (que vamos excluir) e o anterior a ele.
Ou seja, aquele mesmo laço while será usado, da mesma maneira.

Porém, não vamos precisar de um novo nó, afinal não vamos adicionar nada, e sim retirar a struct apontada pelo ponteiro 'atual'.
E como vimos diversas, o ato de 'retirar' um nó de uma lista é simplesmente deixar de apontar algum elemento da lista para ele.

Quem aponta para o elemento que queremos retirar é: anterior->prox
Qual elemento que vem após o elemento que vamos retirar: atual->prox
Elemento retirado, que devemos retornar da função: atual

Ou seja, para excluir, basta apontarmos o elemento anterior ao que queremos retirar, para aquele elemento que vem DEPOIS do que queremos retirar.
Isso é feito da seguinte maneira: anterior->prox = atual->prox

E pronto. A lista continua ligada, mas sem o elemento 'atual', na qual retornamos, sem antes decrementar a variável ***tam***, já que retiramos uma estrutura da lista.

- **Código completo de uma Lista em C**

E finalmente, após muito estudo e trabalho, nosso código completo, de uma lista em C que nos permite inserir e retirar um nó (struct) de toda e qualquer posição.

É uma lista flexível, onde há diversas maneiras de trabalhar com ela, além de exibir seus elementos de uma maneira esteticamente agradável e faz uso de pouca memória (além de liberar ela, ao final da aplicação), sendo robusta e muito e eficiente:

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

struct Node{
	int num;
	struct Node *prox;
};
typedef struct Node node;

int tam;

void inicia(node *LISTA);
int menu(void);
void opcao(node *LISTA, int op);
node *criaNo();
void insereFim(node *LISTA);
void insereInicio(node *LISTA);
void exibe(node *LISTA);
void libera(node *LISTA);
void insere (node *LISTA);
node *retiraInicio(node *LISTA);
node *retiraFim(node *LISTA);
node *retira(node *LISTA);


int main(void)
{
	node *LISTA = (node *) malloc(sizeof(node));
```

```
        if(!LISTA){
                printf("Sem memoria disponivel!\n");
                exit(1);
        }else{
        inicia(LISTA);
        int opt;

        do{
                opt=menu();
                opcao(LISTA,opt);
        }while(opt);

        free(LISTA);
        return 0;
        }
}

void inicia(node *LISTA)
{
        LISTA->prox = NULL;
        tam=0;
}

int menu(void)
{
        int opt;

        printf("Escolha a opcao\n");
        printf("0. Sair\n");
        printf("1. Zerar lista\n");
        printf("2. Exibir lista\n");
        printf("3. Adicionar node no inicio\n");
        printf("4. Adicionar node no final\n");
        printf("5. Escolher onde inserir\n");
        printf("6. Retirar do inicio\n");
        printf("7. Retirar do fim\n");
        printf("8. Escolher de onde tirar\n");
        printf("Opcao: "); scanf("%d", &opt);

        return opt;
}

void opcao(node *LISTA, int op)
{
        node *tmp;
```

```
        switch(op){
            case 0:
                libera(LISTA);
                break;

            case 1:
                libera(LISTA);
                inicia(LISTA);
                break;

            case 2:
                exibe(LISTA);
                break;

            case 3:
                insereInicio(LISTA);
                break;

            case 4:
                insereFim(LISTA);
                break;

            case 5:
                insere(LISTA);
                break;

            case 6:
                tmp= retiraInicio(LISTA);
                printf("Retirado: %3d\n\n", tmp->num);
                break;

            case 7:
                tmp= retiraFim(LISTA);
                printf("Retirado: %3d\n\n", tmp->num);
                break;

            case 8:
                tmp= retira(LISTA);
                printf("Retirado: %3d\n\n", tmp->num);
                break;

            default:
                printf("Comando invalido\n\n");
        }
}
```

```c
int vazia(node *LISTA)
{
	if(LISTA->prox == NULL)
		return 1;
	else
		return 0;
}

node *aloca()
{
	node *novo=(node *) malloc(sizeof(node));
	if(!novo){
		printf("Sem memoria disponivel!\n");
		exit(1);
	}else{
		printf("Novo elemento: "); scanf("%d", &novo->num);
		return novo;
	}
}


void insereFim(node *LISTA)
{
	node *novo=aloca();
	novo->prox = NULL;

	if(vazia(LISTA))
		LISTA->prox=novo;
	else{
		node *tmp = LISTA->prox;

		while(tmp->prox != NULL)
			tmp = tmp->prox;

		tmp->prox = novo;
	}
	tam++;
}

void insereInicio(node *LISTA)
{
	node *novo=aloca();
	node *oldHead = LISTA->prox;
```

```
        LISTA->prox = novo;
        novo->prox = oldHead;

        tam++;
}

void exibe(node *LISTA)
{
        system("clear");
        if(vazia(LISTA)){
                printf("Lista vazia!\n\n");
                return ;
        }

        node *tmp;
        tmp = LISTA->prox;
        printf("Lista:");
        while( tmp != NULL){
                printf("%5d", tmp->num);
                tmp = tmp->prox;
        }
        printf("\n        ");
        int count;
        for(count=0 ; count < tam ; count++)
                printf("  ^  ");
        printf("\nOrdem:");
        for(count=0 ; count < tam ; count++)
                printf("%5d", count+1);


        printf("\n\n");
}

void libera(node *LISTA)
{
        if(!vazia(LISTA)){
                node *proxNode,
                        *atual;

                atual = LISTA->prox;
                while(atual != NULL){
                        proxNode = atual->prox;
                        free(atual);
                        atual = proxNode;
                }
```

```
        }
}

void insere(node *LISTA)
{
        int pos,
                count;
        printf("Em que posicao, [de 1 ate %d] voce deseja inserir: ", tam);
        scanf("%d", &pos);

        if(pos>0 && pos <= tam){
                if(pos==1)
                        insereInicio(LISTA);
                else{
                        node *atual = LISTA->prox,
                                *anterior=LISTA;
                        node *novo=aloca();

                        for(count=1 ; count < pos ; count++){
                                        anterior=atual;
                                        atual=atual->prox;
                        }
                        anterior->prox=novo;
                        novo->prox = atual;
                        tam++;
                }

        }else
                printf("Elemento invalido\n\n");
}

node *retiraInicio(node *LISTA)
{
        if(LISTA->prox == NULL){
                printf("Lista ja esta vazia\n");
                return NULL;
        }else{
                node *tmp = LISTA->prox;
                LISTA->prox = tmp->prox;
                tam--;
                return tmp;
        }

}
```

```
node *retiraFim(node *LISTA)
{
	if(LISTA->prox == NULL){
		printf("Lista ja vazia\n\n");
		return NULL;
	}else{
		node *ultimo = LISTA->prox,
		     *penultimo = LISTA;

		while(ultimo->prox != NULL){
			penultimo = ultimo;
			ultimo = ultimo->prox;
		}

		penultimo->prox = NULL;
		tam--;
		return ultimo;
	}
}

node *retira(node *LISTA)
{
	int opt,
		count;
	printf("Que posicao, [de 1 ate %d] voce deseja retirar: ", tam);
	scanf("%d", &opt);

	if(opt>0 && opt <= tam){
		if(opt==1)
			return retiraInicio(LISTA);
		else{
			node *atual = LISTA->prox,
			     *anterior=LISTA;

			for(count=1 ; count < opt ; count++){
				anterior=atual;
				atual=atual->prox;
			}

		anterior->prox=atual->prox;
		tam--;
		return atual;
		}

	}else{
```

```
            printf("Elemento invalido\n\n");
            return NULL;
        }
}
```

Pilhas (Stack) em C - O Que é e Como Implementar

Neste Tutorial de C, vamos falar sobre uma das estrutura de dados mais importantes da computação, que são as pilhas (*stack*, em inglês).

Vamos entender o que são pilhas, para que servem, como implementar e programar uma, do zero.

Pilha em C - O Que É e Para Que Serve

Uma pilha é um tipo de estrutura de dado, que é uma maneira de se organizar e usar dados, informações.

A regra das pilhas é famoso *LIFO - Last In, First Out*, ou seja, o último a entrar é o primeiro a sair da estrutura.

O nome pilha não é por acaso. Vamos imaginar uma pilha de pratos. Sempre que alguém termina de comer, coloca o prato na pilha, sempre acima.
Ou seja, o último prato a entrar na estrutura, esta sempre no topo.

E na hora de desfazer essa estrutura, essa pilha, que elementos vamos tirar? Sempre o prato de cima, que foi o último elemento a entrar na pilha.
Note que o primeiro prato inserido vai ser o último a sair, pois ele está na base da pilha.

A ideia por trás das pilhas (*stack*), é de suma importância na estrutura de dados, sendo sua lógica usada em uma infinidade de aplicações. Seu sistema operacional, em níveis de linguagem C e Assembly, está a todo momento usando informações em pilhas para tratar processos e chamadas à funções.

Vamos aprender como fazer uma, na mão, do zero?

- **Como Programar Uma Pilha em C**

Aqui, vamos explicando, passo a passo, como criar uma **pilha em C**, do zero, na mão.
Inicialmente vamos comentar tudo que é necessário, toda a lógica, e o código será exibido na próxima parte deste tutorial.

Struct e Cabeçalhos de Funções

O primeiro passo é a struct, vamos de Node (cada elemento de uma pilha é chamado de nó).
Ela vai ter apenas dois elementos, um número inteiro e um ponteiro para outra struct do tipo Node.

Esse ponteiro do próprio tipo é obrigatório, não existe estrutura dinâmica de dados sem esse detalhe especial. Na lógica do funcionamento da pilha, vamos entender para que ele serve.

Já o outro elemento (**int num**) é só para armazenarmos números nessa pilha, pois vamos pedir e mostrar esses números. Mas note que isso é uma struct, podemos colocar quantos elementos e do tamanho que quisermos, que a pilha vai funcionar do mesmo jeito.

Também colocamos todos os cabeçalhos das funções que iremos usar para programar a pilha em C, por questões de organização (o código das funções ficará abaixo).

- Função main()

Na função main criamos a nossa pilha, que é uma struct Node, vamos chamar ela de "PILHA", e é a base. Quando o primeiro elemento for adicionado, ele vai ser adicionado no ponteiro "prox" da PILHA.
Este nó, na verdade, não faz parte da pilha, ele serve apenas para indicar onde ela começa (começa no ponteiro na qual ela aponta).

O resto da main é simplesmente um laço do while, que fica exibindo um menu com as opções para trabalharmos com a pilha, que termina se digitarmos 0.

- Função menu() e opcao()

Essas são as funções responsáveis pela ação, a interação o usuário e a pilha.

A função menu() simplesmente exibe as opções possíveis e pede um inteiro ao usuário.

Este inteiro será usado e passado para a função opcao(), que junto com a pilha (ponteiro *PILHA) vai servir para chamar a função específica, de acordo com o que o usuário escolheu.

Na função opcao(), basicamente existe um switch que vai tratar a opção escolhida pela usuário, e chamar a função correta. Sem segredo.

- Função inicia()

Esta função é responsável por inicializar a pilha.
Inicializar é simplesmente preparar a pilha para ser utilizada, simplesmente apontando seu ponteiro *prox para NULL.

Essa função é chamada automaticamente no início de nosso programa em C, e quando zeramos a pilha.

- Função vazia()

Esta função simplesmente checa se a pilha está vazia ou não.
Basta olhar para onde aponta a base (*PILHA), se apontar para NULL é porque ela está vazia, senão, é porque existem nós nesta estrutura de dados dinâmica.

- Função aloca()

Visando deixar nosso algoritmo bem feito, estruturado e organizado, é interessante separar cada ideia em uma função diferente, com um propósito bem evidente e único.

A função aloca() é um exemplo dessa organização.
Como o nome sugere, ele serve para alocar nós.
Sempre que formos adicionar um elemento na pilha, temos que alocar memória para ele.

Essa função aloca a memória necessária pro nó (struct Node), pede o número que o usuário quer armazenar) e retornar o endereço da memória alocada.

- Função libera()

Tão importante quanto alocar memória para cada nó da pilha de nossa estrutura de dados, é liberar esse espaço de memória. A função libera faz isso, vai liberando o espaço alocado de cada nó de nossa pilha.

Usamos dois ponteiros para a struct do nó, o ponteiro que aponta para o elemento atual e o ponteiro que aponta para o próximo elemento.
Pegamos o ponteiro que aponta para o nó atual e usamos para desalocar aquele espaço de memória, em seguida o ponteiro que apontava para o atual aponta para o próximo, e isso segue até o fim da pilha, desalocando cada um dos nós da estrutura de dados.

- Função exibe()

Essa é a função responsável por exibir todos os elementos da pilha.
Como em cada nó dessa estrutura de dados possui um inteiro que o usuário inseriu, essa função, no fim das contas, vai exibir os números da pilha.

Essa função declara um ponteiro que vai começar apontando para o primeiro elemento da pilha, exibe o número armazenado ali, pega o endereço do próximo nó, exibe o que está armazenado nele também, e assim se segue, até o fim da pilha (quando *prox aponta para NULL).

- Função push()

Agora vamos a parte que mais interessa em se tratando de estrutura de dados, e especificamente, sobre pilhas em C: as funções push e pop.

Push em inglês é empurrar, vamos empurrar, colocar um elemento, um nó na pilha.
O primeiro passo é alocar espaço para esteve novo nó da pilha, o que é feito com ajuda da função aloca().
Como é uma pilha, seu último elemento (que é esteve novo), deve apontar para NULL, pois isso caracteriza o fim da pilha.

Adicionado o elemento, vamos procurar o último elemento da pilha.
Temos o ponteiro *PILHA que aponta para a base.
Se a pilha estiver vazia, ótimo! Fazemos o ponteiro *prox apontar para esteve novo nó, e tudo ok.

Se a pilha não for vazia, vamos achar o último elemento através de um ponteiro *tmp que vai apontar para o primeiro elemento da pilha (PILHA->prox aponta para o primeiro nó).

E como sabemos que o nó atual é o último?
Basta checar seu ponteiro *prox, se ele apontar para NULL, ele é último.
Se não apontar, é porque aponta para um novo nó, então fazemos nosso ponteiro *tmp apontar para este novo nó, sempre, até chegar no último.

Quando "tmp->prox" apontar para NULL, é porque *tmp está apontando para o último nó.
Agora, vamos fazer o próximo nó apontar para nosso novo nó: tmp->prox = novo
E pronto! Função push feita! Adicionamos um novo nó na pilha.

- Função pop()

Agora vamos para a função pop, o outro pilar da estrutura de dados dinâmica que é a pilha.
Esta função vai tirar o último nó da pilha, e retirá-lo de lá.

Primeiro fazemos uma checagem se a pilha está vazia (PILHA->prox aponta pra NULL).
Se estiver, não há nada a ser feito, pois não há nó para ser retirado da pilha.

Do contrário, vamos utilizar dois ponteiros para struct Node, o "ultimo" e o "penultimo".
Basicamente, o que vamos fazer é que o "ultimo" aponte para o último elemento da pilha e o "penultimo" aponte para o último nó da pilha.

O motivo disso é simples: vamos retornar o último nó da pilha e vamos retirá-lo da lista (então ele vai se perder, por isso precisaremos sempre do penúltimo, pois este vai se tornar o novo último nó da lista).

O que vamos fazer é buscar o último nó (que é aquele que tem o ponteiro *prox apontando pra NULL).

E sempre que avançarmos na pilha com o ponteiro "ultimo", fazemos com que o "penultimo" também avance (ora, o penúltimo nó é aquele que tem o ponteiro *prox apontando para o ponteiro *ultimo).

Essa lógica é feita testando ultima->prox, quando não for NULL, o ponteiro "penultimo" passa a ser o "ultimo" e o "ultimo" vai ser o "ultimo->prox", que é o próximo nó da pilha.
Note que agora que demos um passo a frente na pilha, com os dois ponteiros.

E isso só para quando estivermos apontando para o último e penúltimo nó da pilha.

Quando estivermos nesse ponto, fazemos "penultimo->prox" apontar para NULL, pois vai caracterizar que o penúltimo nó será, agora, o último nó (pois aponta pra NULL), ou seja: retiramos o último nó da pilha!

E o que fazemos com o último nó?
Vamos retornar ele! Se estamos tirando ele da pilha, é porque queremos o que tem nele, seja lá pra que for. Então retornamos ele pra função que o chamou (a função opcao()), ela exibe o valor desse último elemento da pilha e então o descarta (liberando a memória dele).

Código Completo De Uma Pilha em C

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

struct Node{
 int num;
 struct Node *prox;
};
typedef struct Node node;

int tam;

int menu(void);
void inicia(node *PILHA);
```

```c
void opcao(node *PILHA, int op);
void exibe(node *PILHA);
void libera(node *PILHA);
void push(node *PILHA);
node *pop(node *PILHA);


int main(void)
{
 node *PILHA = (node *) malloc(sizeof(node));
 if(!PILHA){
  printf("Sem memoria disponivel!\n");
  exit(1);
 }else{
 inicia(PILHA);
 int opt;

 do{
  opt=menu();
  opcao(PILHA,opt);
 }while(opt);

 free(PILHA);
 return 0;
 }
}

void inicia(node *PILHA)
{
 PILHA->prox = NULL;
 tam=0;
}

int menu(void)
{
 int opt;

 printf("Escolha a opcao\n");
 printf("0. Sair\n");
 printf("1. Zerar PILHA\n");
 printf("2. Exibir PILHA\n");
 printf("3. PUSH\n");
 printf("4. POP\n");
 printf("Opcao: "); scanf("%d", &opt);
```

```
 return opt;
}

void opcao(node *PILHA, int op)
{
 node *tmp;
 switch(op){
  case 0:
   libera(PILHA);
   break;

  case 1:
   libera(PILHA);
   inicia(PILHA);
   break;

  case 2:
   exibe(PILHA);
   break;

  case 3:
   push(PILHA);
   break;

  case 4:
   tmp= pop(PILHA);
   if(tmp != NULL)
   printf("Retirado: %3d\n\n", tmp->num);
   break;

  default:
   printf("Comando invalido\n\n");
 }
}

int vazia(node *PILHA)
{
 if(PILHA->prox == NULL)
  return 1;
 else
  return 0;
}

node *aloca()
{
```

```
 node *novo=(node *) malloc(sizeof(node));
 if(!novo){
  printf("Sem memoria disponivel!\n");
  exit(1);
 }else{
  printf("Novo elemento: "); scanf("%d", &novo->num);
  return novo;
 }
}


void exibe(node *PILHA)
{
 if(vazia(PILHA)){
  printf("PILHA vazia!\n\n");
  return ;
 }

 node *tmp;
 tmp = PILHA->prox;
 printf("PILHA:");
 while( tmp != NULL){
  printf("%5d", tmp->num);
  tmp = tmp->prox;
 }
 printf("\n        ");
 int count;
 for(count=0 ; count < tam ; count++)
  printf("  ^  ");
 printf("\nOrdem:");
 for(count=0 ; count < tam ; count++)
  printf("%5d", count+1);


 printf("\n\n");
}

void libera(node *PILHA)
{
 if(!vazia(PILHA)){
  node *proxNode,
     *atual;

  atual = PILHA->prox;
  while(atual != NULL){
```

```
    proxNode = atual->prox;
    free(atual);
    atual = proxNode;
   }
  }
}

void push(node *PILHA)
{
 node *novo=aloca();
 novo->prox = NULL;

 if(vazia(PILHA))
  PILHA->prox=novo;
 else{
  node *tmp = PILHA->prox;

  while(tmp->prox != NULL)
   tmp = tmp->prox;

  tmp->prox = novo;
 }
 tam++;
}


node *pop(node *PILHA)
{
 if(PILHA->prox == NULL){
  printf("PILHA ja vazia\n\n");
  return NULL;
 }else{
  node *ultimo = PILHA->prox,
            *penultimo = PILHA;

  while(ultimo->prox != NULL){
   penultimo = ultimo;
   ultimo = ultimo->prox;
  }

  penultimo->prox = NULL;
  tam--;
  return ultimo;
 }
}
```

- **Considerações sobre a Estrutura de Dados Dinâmica Pilha**

A ideia por trás da estrutura de dados do tipo pilha é simples: o último elemento a entrar, é sempre o primeiro a sair.

Existem zilhões de maneiras de se programar uma pilha.
Fizemos apenas uma, que criamos e achamos que é didaticamente interessante de se aprender

Por exemplo, fizemos uma que possui apenas um elemento fixo, o ponteiro *PILHA, que aponta para a base da pilha. Tente fazer um com o ponteiro *top, que aponta para o topo (final da pilha).

Se tiver estudado nosso Tutorial Sobre Listas Encadeadas em C vai entender bem melhor tudo que foi explicado neste tutorial, e vai notar que uma pilha é uma lista onde os só podemos inserir elementos no fim da lista e só podemos retirar elementos do fim também.

Ou seja, uma pilha é uma lista também, mas uma lista mais simples, onde só usamos o final dela.

Mais importante que ter todos esses códigos e implementações em C em mente, é ter essas ideias, essa lógica da coisa, de como funciona.

Filas em C - Como Programar

Neste Tutorial de C de nossa apostila online, vamos falar sobre uma importante estrutura de dados dinâmica, que é a fila.

Junto com as listas encadeadas e as pilhas em C, formam o conjunto de estrutura de dados mais importantes e usados.

- **Filas em C - Tutorial de Queue**

As filas (*queue*, em inglês) são um tipo de estrutura dinâmica de dados onde os elementos (ou nós) estão arranjados em lista que obedece as seguintes regras:
- Ao inserir um nó, ele vai para a última posição da estrutura
- Ao retirar um nó, é tirado o primeiro elemento da estrutura

Este tipo de estrutura de dados é dita ser *FIFO* (*First in, first out*), ou seja, o primeiro elemento a entrar na estrutura é o primeiro a sair.

O nome **fila**, por si só, já é auto-explicativo.
Imagine uma fila de banco. A primeira pessoa que chegou na fila, é a que vai ser atendida primeiro.

E se chegar mais alguém? Ela vai demorar mais pra ser atendida.
E se chegar uma outra? Ela será a última a ser atendida, pois quem está na sua frente é atendido antes.

Ou seja, sempre que inserimos elementos nessa fila, inserimos ao final.
E sempre que retiramos, estamos tirando o primeiro elemento da fila (o mais antigo), pois o que está na frente que vai sair antes.

Resumindo: inserimos ao fim, e retiramos do começo.
Diferente das pilhas em C (stack), onde colocamos no fim e retiramos do fim.

Simples, não?
Vamos entender agora a lógica para programar uma fila em C, do zero.

- **Filas em C - Como Programar a Estrutura de Dados**

Função main(), opcao() e menu()

Ok, agora que já sabemos o que é uma fila e como funciona, vamos começar a implementar essa estrutura de dados em C, partindo do zero, apenas baseado no que sabemos da definição de FIFO.

Inicialmente criamos uma struct chamada Node, que é vai criar os nós de nossa estrutura de dados.
Vamos colocar nela um inteiro, o "num" para armazenar um valor qualquer inserido pelo usuário.
Obrigatoriamente, temos que inserir nessa struct um ponteiro de seu próprio tipo.

Vamos chamar esse ponteiro de "prox" pois ele vai servir para apontar para o próximo nó da estrutura.
Caso esse ponteiro aponte para NULL, é porque seu nó é o último da fila ou a fila está vazia.

De resto, a função principal **main()** simplesmente fica chamando (em um looping do while) a função menu(), que simplesmente exibe as opções possíveis para se trabalhar com a fila em C.

Depois que o usuário escolhe o que fazer, sua escolha é enviada para a função **opcao()**, que usa um teste condicional do tipo switch para invocar a função correta.

Função aloca() e inicia()

Inicialmente, criamos um ponteiro para a struct Node, é o "FILA", e ele é a representação de nossa estrutura de dados do tipo fila.
Seu inteiro não importa, a função deste ponteiro é definida pelo seu ponteiro "prox".

O vetor "prox" aponta pro primeiro elemento da fila, e caso ele não exista, a fila está vazia e "prox" aponta para NULL.
Fazemos isso na função **inicia()**, que inicializa a fila fazendo: FILA->prox = NULL

Devemos sempre lembrar de alocar memória para cada nó de nossa fila, o que é feito pela função **aloca()**.

Função vazia()

No decorrer de nosso programa sobre estrutura de dados do tipo fila, muitas vezes será necessário checar se a fila está vazia ou não.

Isso é feito de uma maneira bem simples: checando o ponteiro "prox" da struct "FILA".
Se apontar pra NULL, a fila está vazia. Do contrário, tem pelo menos um elemento na fila.

Função insere()

Antes de mais nada, vamos usar a função aloca() para reservar um espaço em memória para o novo nó, o "novo". Como este novo nó será o último da fila, seu ponteiro "prox" deve apontar para NULL.
Esta foi a primeira parte do processo de se adicionar um elemento em uma fila.

A segunda parte é adicionar este novo nó ao final da fila, e para tal, devemos achar o último nó da fila.
Primeiro checamos se a fila está vazia, pois se tiver, basta colocar no novo nó em FILA->prox

Caso não esteja, criamos um ponteiro "tmp" que vai apontar para todos os elementos da fila em busca do último. Ele começa no primeiro elemento, que está em "FILA->prox".
Se "tmp->prox" apontar para NULL, o ponteiro aponta para o último da fila.
Senão, devemos seguir adiante com o ponteiro (tmp = tmp->prox) até acharmos o último elemento.

Achando, colocamos lá o novo nó, o "novo".
A variável inteira "tam" é para definir o tamanho da fila (número de nós).
Usaremos este inteiro na função "exibe()", que vai exibir os elementos da fila.

Função retira()

Vamos agora retirar um elemento da fila.

E segunda a lógica deste tipo de estrutura de dados, vamos retirar o primeiro nó.

Antes de tudo, checamos se a fila não está vazia.
Se estiver, trabalho feito, pois não precisaremos retirar nó algum da estrutura de dados.

Caso a fila não esteja vazia, precisamos identificar o primeiro elemento e o segundo (na verdade, não é obrigado que exista um segundo elemento). O que precisamos fazer é que "FILA->prox" não aponte mais para o primeiro elemento, e sim para o segundo.

Vamos usar um ponteiro "tmp" para apontar para o primeiro elemento da fila:
tmp = FILA->prox
Se "tmp" aponta para o primeiro elemento, então "tmp->prox" aponta para o segundo elemento ou NULL, caso a fila só tenha um nó.

Agora vamos fazer a ligação entre o ponteiro base (FILA) e o segundo elemento (ou NULL) da fila:
FILA->prox = tmp->prox

Pronto. Tiramos o primeiro elemento da jogada, pois se ninguém aponta para ele, ele não faz mais parte da estrutura de dados.
Interessante, não?

Note que declaramos a função "retira()" como sendo do tipo struct Node, pois é uma boa prática retornar o nó que retiramos, pois geralmente retiramos ele da estrutura para fazer algo, trabalhar em cima dele. Depois que retornamos ele pra função "opcao()" liberamos o espaço que havia sido alocado para ele.

Função exibe()

Esta função serve somente para mostrar os números ("num") existentes em cada nó, para você poder adicionar, retirar e ver o que vai acontecendo com a fila.

Ou seja, esta função tem mais propósitos didáticos, pra você ver as coisas realmente funcionamento na sua frente, como devem funcionar.

Basicamente pegamos um ponteiro "tmp" e fazemos ele apontar para cada um dos nós, exibindo o número. Para saber o tanto de nós existentes e a ordem, usamos a variável "tam" que é incrementada quando adicionamos nó na fila (função insere) e decrementada quando tiramos elementos da estrutura de dados (função retira).

Função libera()

Esta função simplesmente tem por objetivo ir em cada nó e liberar a memória alocada.
Para tal, usamos dois ponteiros.

Um ponteiro aponta para um nó ("atual"), e o outro ponteiro aponta para o nó seguinte ("proxNode").
Liberamos o primeiro ponteiro, ou seja, aquele nó deixa de existir.
Se tivéssemos só este ponteiro, nos perderíamos na fila.

Porém, o ponteiro que aponta para o nó seguinte da fila serve pra isso, pois agora temos o ponteiro para o próximo elemento da fila "proxNode").
Agora damos um passo pra frente, fazendo um ponteiro apontar para este próximo elemento ("atual = proxNode"), e o ponteiro próximo, para uma posição a frente ("proxNode = proxNode->prox").

E repetimos o processo até que o nó atual seja NULL (fim da fila).

- **Fila em C - Código Fonte Completo**

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

struct Node{
	int num;
	struct Node *prox;
};
typedef struct Node node;

int tam;

int menu(void);
void opcao(node *FILA, int op);
void inicia(node *FILA);
int vazia(node *FILA);
```

```
node *aloca();
void insere(node *FILA);
node *retira(node *FILA);
void exibe(node *FILA);
void libera(node *FILA);


int main(void)
{
	node *FILA = (node *) malloc(sizeof(node));
	if(!FILA){
		printf("Sem memoria disponivel!\n");
		exit(1);
	}else{
	inicia(FILA);
	int opt;

	do{
		opt=menu();
		opcao(FILA,opt);
	}while(opt);

	free(FILA);
	return 0;
	}
}


int menu(void)
{
	int opt;

	printf("Escolha a opcao\n");
	printf("0. Sair\n");
	printf("1. Zerar fila\n");
	printf("2. Exibir fila\n");
	printf("3. Adicionar Elemento na Fila\n");
	printf("4. Retirar Elemento da Fila\n");
	printf("Opcao: "); scanf("%d", &opt);

	return opt;
}

void opcao(node *FILA, int op)
{
```

```
        node *tmp;
        switch(op){
                case 0:
                        libera(FILA);
                        break;

                case 1:
                        libera(FILA);
                        inicia(FILA);
                        break;

                case 2:
                        exibe(FILA);
                        break;

                case 3:
                        insere(FILA);
                        break;

                case 4:
                        tmp= retira(FILA);
                        if(tmp != NULL){
                                printf("Retirado: %3d\n\n", tmp->num);
                                libera(tmp);
                        }
                        break;

                default:
                        printf("Comando invalido\n\n");
        }
}

void inicia(node *FILA)
{
        FILA->prox = NULL;
        tam=0;
}

int vazia(node *FILA)
{
        if(FILA->prox == NULL)
                return 1;
        else
                return 0;
}
```

```
node *aloca()
{
	node *novo=(node *) malloc(sizeof(node));
	if(!novo){
		printf("Sem memoria disponivel!\n");
		exit(1);
	}else{
		printf("Novo elemento: "); scanf("%d", &novo->num);
		return novo;
	}
}

void insere(node *FILA)
{
	node *novo=aloca();
	novo->prox = NULL;

	if(vazia(FILA))
		FILA->prox=novo;
	else{
		node *tmp = FILA->prox;

		while(tmp->prox != NULL)
			tmp = tmp->prox;

		tmp->prox = novo;
	}
	tam++;
}


node *retira(node *FILA)
{
	if(FILA->prox == NULL){
		printf("Fila ja esta vazia\n");
		return NULL;
	}else{
		node *tmp = FILA->prox;
		FILA->prox = tmp->prox;
		tam--;
		return tmp;
	}

}
```

```c
void exibe(node *FILA)
{
	if(vazia(FILA)){
		printf("Fila vazia!\n\n");
		return ;
	}

	node *tmp;
	tmp = FILA->prox;
	printf("Fila :");
	while( tmp != NULL){
		printf("%5d", tmp->num);
		tmp = tmp->prox;
	}
	printf("\n        ");
	int count;
	for(count=0 ; count < tam ; count++)
		printf("  ^  ");
	printf("\nOrdem:");
	for(count=0 ; count < tam ; count++)
		printf("%5d", count+1);


	printf("\n\n");
}

void libera(node *FILA)
{
	if(!vazia(FILA)){
		node *proxNode,
		     *atual;

		atual = FILA->prox;
		while(atual != NULL){
			proxNode = atual->prox;
			free(atual);
			atual = proxNode;
		}
	}
}
```

Arquivos

Até o momento em nosso curso, todas as variáveis, cálculos e dados de nossos programas foram temporários.

Isto é, ao fechar o programa, tudo se perdia. E ao iniciar nossos aplicativos, tudo começava do zero novamente. Mas na prática, poucos programas funcionam assim.

A gigantesca maioria dos programas que usamos armazenam (salvam) informações em algum lugar (HD, pendrive, banco de dados de um servidor etc) e consultam informações de algum local.

Essa interação entre um programa e um sistema é feita através de arquivos: arquivos de texto, de imagem, de música, de vídeo, arquivos criptografados etc.

E é isto que iremos aprender nesta seção de nossa **apostila de C**. Vamos aprender a armazenar informações diretamente de nossos programas, bem como consultar dados armazenados em qualquer local de nosso disco rígido.

Arquivos (Files) em C

Neste tutorial de nossa **apostila de C**, daremos início a uma importante seção de nosso curso, sobre os arquivos (*Files*), em linguagem C.

- **Dados temporários x Dados permanentes**

Se notar bem, até o momento, em nossa apostila, todos os nossos programas iniciavam sempre do mesmo jeito, em relação aos dados.

Quando abríamos, informávamos variáveis diversas, como números e strings.

Eles executam comandos que programamos e, ao fecharem, essas informações eram todas perdidas em nossa máquina.
E ao iniciá-los de novo, tudo voltava a rodar do zero.

Resumindo: nossos dados, variáveis e tudo mais, eram temporários.
Basta desligar o programa, sistema operacional ou computador, que tudo que tínhamos feito iria embora.

Mas é assim normalmente?
Quando você inicia seu sistema operacional, ele começa tudo do zero?
Sem variáveis, sem informações?

Claro que não. Suas pastas estão lá, com seus arquivos de texto, música e imagens.

Ou seja, o programas que rodam, de alguma maneira, fazem com que estas informações fiquem registradas em sua máquina de maneira permanente.

Imagine que você implementou uma lista em C, centenas de linhas de código, muitas ideias e algoritmos. Essa lista é usada em uma farmácia para armazenar os dados de todos os remédios, com seus nomes, indicações, contra-indicações, valores, unidades em estoque etc.

Ok, isso é perfeitamente possível de se criar, e o ensinamos com o uso da lista.

Mas será que o dono do estabelecimento iria gostar de colocar todas as informações, sobre todos os medicamentos, sempre que ligasse o que computador?

Óbvio que não. Por isso, precisamos aprender não só como gravar informações de maneira permanente, mas também ler, alterar e apagá-las.

- **Arquivos em C - O que são e para que servem**

Já reparou que, quando você vai jogar aquele game que você adora você põe seus dados, aí joga de boas, e quando vai abrir ele de novo vê que ele guardou seu nome, pontuação e situação do jogo que você estava? Já parou para pensar como isso ocorre?

A resposta é bem simples: eles armazenam essa informação em alguma local de seu computador, em arquivos. Assim, ao iniciar o programa de novo, o jogo simplesmente vai lendo essas informações para saber de onde começar, que dados exibir.

E é exatamente isso que vamos fazer ao estudar arquivos em C.
Iremos aprender como interagir com o sistema, como:

- ler arquivos de textos
- escrever em arquivos (ao invés de imprimir na tela, imprimir em arquivos)
- apagar e substituir informações em arquivos
- após rodar nosso aplicativo, salvar nossos valores
- usar dados armazenados em arquivos para usar em nossos programas

Poderemos, por exemplo, criar um jogo que pode salvar nossa pontuação para continuarmos depois.

Assim é possível criar um jogo da Forca, ondas as palavras para serem adivinhadas ficam em arquivos de textos, sendo possível o usuário ir nesses arquivos e adicionar mais palavras para o jogo.

Enfim, é um mundo de novas e interessantes opções que teremos ao fazer uso de arquivos em nossos aplicativos em C.

- **Comandando um Sistema Operacional através do uso de Arquivos**

Escrever, ler, apagar e editar arquivos é um assunto básico e essencial, que praticamente toda linguagem de programação permite. Porém, dependendo de seu sistema operacional, esse recurso pode ser pouco útil ou pode ser que lhe dê todo o poder do mundo.

Em sistemas operacionais derivados do Unix, como o Linux e os *BSD, praticamente tudo é feito usando arquivos de texto, como os .txt que você utiliza.

Quer adicionar um usuário? Pode adicionar através de arquivo de texto.
Quer montar uma rede? Configure alguns textos.
Quer baixar algum aplicativo ou fazer uma atualização? Use uns comandos no terminal, em texto.

Até para definir o brilho de sua tela, é possível fazer simplesmente alterando o conteúdo de arquivos de texto.
E como dissemos, é possível alterar, ler, apagar e escrever em arquivos através da linguagem C.

O que isso nos diz?

Sim, sabendo programar e usando sistemas operacionais, como Linux, é possível revirar o sistema de ponta cabeça, fazendo o que você quiser.
Por isso indicamos que use, pelo menos para experimentar, esse maravilhoso Sistema.

Ao contrário do que muitos pensam, não é difícil, para instalar basta colocar um pendrive ou dvd de uma ISO, e está feito. É possível instalar e usar tudo no modo gráfico.

É um sistema simplesmente perfeito para quem é envolvido com programação.

O sistema já vem com diversos compiladores e IDE's, prontinho para você programar.

Infelizmente tal liberdade não é, nem de longe, possível com sistemas fechados, como o Windows.

Neste sistema, se quiser ajustar ou mudar algo na configuração, provavelmente terá que ir no Painel de Controle e executar algum aplicativo (binário), que obviamente é fechado, não sendo possível usar através de arquivos de texto, até porque não sabemos como é o funcionamento de tais programas.

Mas independente de seus sistema, verá que usar arquivos em seus programas em C os fará bem mais flexíveis e úteis.

Operações com arquivos: Modos de abertura (FILE e fopen), fechamento (EOF, fclose e fcloseall), modo texto e binário

No tutorial passado de nossa **apostila de C**, falamos sobre o que são e a importância dos arquivos em linguagem C, onde mostramos que para termos programas mais robustos e flexíveis, seria necessário fazer uma comunicação entre nossos aplicativos e o sistema.

Essa comunicação é feita através dos arquivos, que servem tanto para fornecer informações para nossos programas, como para salvar dados deles.

Neste artigo vamos entrar em detalhes sobre a abertura, fechamento e outras operações com arquivos.

- **O que são arquivos**

Basicamente, arquivos são dados.
Mas por dados, podemos estar dizendo muita coisa.

No contexto geral, quando falamos em arquivos podemos estar nos referindo por um arquivo de texto, uma imagem, uma música, um programa de edição de imagem ou o executável que você criou estudando pelo nosso curso de C.

Porém, para o computador (leia-se: hardware), é tudo bit.
Tudo é visto como 1s (true, estado alto) ou 0s (false, estado baixo).
Na verdade, quem já estudou eletrônica, sabe que isso de 1 e 0 é apenas uma abstração, o que ocorre de fato é a existência de voltagem baixa (bit 0) e alta (bit 1), ou existência ou não de corrente.

Existem diversas maneiras de se representar um arquivo, além de toda uma estrutura e hierarquia de dados, sendo o sistema de arquivos de um sistema operacional um ramo bem complexo de estudo.

Em nossa apostila, você pode entender o conceito de arquivo apenas como uma série de *bytes*, de informações dispostas de maneira sequencial (*stream*), onde teremos informações e controle sobre seu início e fim.

- **Modo texto e binário**

Como dissemos ao longo de nosso curso, a linguagem C é extremamente poderosa e nos oferece a possibilidade da fazer praticamente qualquer coisa com nossas máquinas (e não só microcomputadores).

Uma dessas capacidades é possibilidade de trabalhar com arquivos no modo binário.
Ou seja, podemos abrir, criar e salvar um arquivo no modo binário.
Por exemplo, é possível fazer uma cópia binária de arquivo. Isto é, pegar bit por bit de um executável e colocar em outro arquivo, duplicando o arquivo.

Porém, hardware é que se dá bem e gosta de trabalhar com bit (quem programa em Assembly também :). Para humanos, uma lista de milhões ou bilhões de números 1 e 0 é algo quase impossível de ser interpretado.

Por isso, os programadores geralmente trabalham com arquivos em modo texto, que será o foco aqui de nossa apostila. Basicamente iremos tratar de arquivos em texto, como os de extensão .txt, que nada mais serão do que uma lista de caracteres (dígitos, letras e símbolos, como espaço e acentuação).

Mais adiante neste tutorial, iremos explicar em detalhes como trabalhar com arquivos em modo texto ou modo binário.

- **Como abrir um arquivo: A função fopen() e o ponteiro FILE**

Para abrir, alterar ou criar um arquivo em seu HD é necessário criar uma espécie de vínculo entre seu programa e um determinado endereço de memória, onde está localizado (ou vai se localizar) o arquivo.

Essa ligação é um pouco mais complexa e de 'baixo nível', mas isso não será motivo de preocupação, pois na linguagem C existe um tipo de dado, o ***FILE***, que serve para trabalharmos especificamente com arquivos, e já está inclusive na biblioteca padrão do C, assim, podemos usar sem nos preocupar em incluir biblioteca ou criar algum código.

Essa ligação, entre nosso programa e seu disco de dados é feita através de um ponteiro.

Tal ponteiro é do tipo ***FILE***, e o nome você escolhe.
A sintaxe é a de criação de um ponteiro:
FILE *arquivo;

Assim, nova variável *arquivo*, do tipo *FILE*, será a relação entre nosso programa em C e o arquivo que iremos trabalhar.
Porém, sabemos que o ponteiro deve apontar para um tipo específico de dados, que no nosso caso é o tipo ***FILE***.

Mas ele precisa apontar para 'alguém', só ter a referência (endereço de memória onde o ponteiro aponta) não serve de muita coisa. E é aí que entra a função ***fopen()***.

A função *fopen()* serve para especificarmos algumas características do arquivo.
Especificamente, como pode ser visto na declaração desta função, precisamos fornecer dois dados:
Endereço do arquivo no seu computador (representado por uma string)
Modo de abertura do arquivo (outra string contendo o método de abertura)

O protótipo dessa função é:
FILE *fopen(const char *filename, const char *mode)

- **Nome de um arquivo**

Para associar nossa variável ponteiro para um arquivo que será lido ou criado em seu computador, é necessário que você forneça a localização desse arquivo, afinal o C não pode adivinhar isso.

Isso é feito através de uma string passada para a função *fopen()*, e essa string deve conter o endereço completo do arquivo, e isso inclui o nome do arquivo.
Vamos supor que vamos trabalhar com um arquivo chamado "*arquivo.txt*".

Se queremos usar este arquivo e ele está na MESMA PASTA (ou diretório) do seu executável, precisamos apenas fornecer o nome completo (com extensão) do arquivo:
"*arquivo.txt*"

E caso ele esteja em outra pasta, como no C: ?
Fornecemos o caminho inteiro, que é C:\arquivo.txt

Porém, aqui vai um detalhe que é um erro bem comum entre iniciantes: para representar a barra "\" em uma string, usamos DUAS BARRAS: \\

Assim, nossa string seria: "C:\\arquivo.txt"
Se o arquivo estiver numa pasta (vamos chamar de "meus_arquivos", dentro da pasta que está o executável, fazemos: "meus_arquivos\\arquivo.txt"

Ou podemos pedir o endereço para o usuário, que poderá escrever usando apenas uma barra "\", pois a representação dela em string é "\\" e o C faz essa interpretação automaticamente.

- **Modos de abertura: read (r), write (w) e append(a)**

read(r) - Leitura de arquivo

r

Sempre que quisermos ler um arquivo, usamos o modo *read*, que será representado simplesmente pela letra "r":
FILE *arquivo = fopen("arquivo.txt", "r");

Isso faz o arquivo ser aberto unicamente para ser lido (não podemos alterar nada).

r+

Se adicionarmos um sinal de "+" no "r", será possível abrir o arquivo tanto para leitura como para escrita, e caso ele não exista, será automaticamente criado.
E caso exista, o seu conteúdo anterior será apagado e substituído pelo novo:
FILE *arquivo = fopen("arquivo.txt", "r+")

rb

Abre o arquivo para leitura, mas em modo binário:
FILE *arquivo = fopen("arquivo.txt", "rb")

write(w) - Escrita em arquivo

w

Para abrirmos um arquivo para escrita (colocar informações do nosso programa no arquivo), usamos a letra "w". Esse modo automaticamente cria o arquivo, ou substitui seu conteúdo anterior:
FILE *arquivo = fopen("arquivo.txt", "W")

w+

Abertura do arquivo tanto para leitura como para escrita.
Caso já exista o arquivo, seu conteúdo será substituído:
FILE *arquivo = fopen("arquivo.txt", "w+")

wb

Escrita em arquivos no modo binário:
FILE *arquivo = fopen("arquivo.txt", "wb")

append(a) - Escrevendo ao final do arquivo (anexando)

a

Ao usar o "w" para escrever (*write*), começamos a inserir informação no início do arquivo.
Mas, e se quisermos inserir ao final? Se não quisermos apagar o que tinha lá antes?

Para isso vamos ANEXAR, informações! Anexar em inglês é *append*, e fazemos isso abrindo o arquivo usando o modo de abertura "a":
FILE *arquivo = fopen("arquivo.txt", "a")

a+

Caso queiramos abrir um arquivo para leitura ou para escrita ao FINAL do arquivo (anexar), usamos o símbolo "+" depois da letra "a":
FILE *arquivo = fopen("arquivo.txt", "a+")

ab

Por afim, assim como na leitura "rb" e escrita binária "wb", podemos anexar informações ao final do arquivo, de maneira binária usando o "ab":

```
FILE *arquivo = fopen("arquivo.txt", "ab")
```

- **EOF (*end of file*), final do arquivo - Fechando arquivos com *fclose()* e *fcloseall()***

Para saber o final do arquivo (que para o sistema, é uma sequência de bytes), o C vai procurar um sinal, uma constante conhecida por ***EOF***, que sinaliza o final do arquivo.

Para identificar o final de um arquivo, podemos usar a função *fclose*, que recebe um ponteiro para o tipo ***FILE*** e retorna um inteiro:

```
int fclose(FILE *arq)
```

Caso o byte lido represente o ***EOF***, a função "fecha" a abertura do arquivo. Ou seja, libera a memória associado ao ponteiro do ***FILE****.

Assim como ensinamos em ponteiros, ao usar a função free() para liberar memória alocada, fechar os arquivos que você não está mais usando é uma excelente prática de programação.

E caso trabalhe com diversos arquivos, você poderá fechá-los todos de uma vez, através da função:

```
int fcloseall()
```

- **Erros em abertura de arquivos**

Quando abrimos um arquivo, o ponteiro que criamos do tipo ***FILE*** armazenará o endereço de um arquivo.

Porém, nem sempre esta tarefa é possível, gerando um erro.
Quando este erro ocorre o ponteiro irá apontar para NULL, sendo essa prática (checagem se o ponteiro aponta para NULL) muito importante para o tratamento de erros na abertura de **arquivos em C**.

Este erro pode ocorrer por vários motivos. O mais óbvio é abrir um arquivo que não existe (geralmente ocorre quando escrevemos errado o endereço do arquivo).

Em sistemas operacionais mais seguros, como os Linux, o acesso aos arquivos (seja para leitura, escrita ou anexação) é geralmente limitado, não

sendo possível acessar e alterar qualquer arquivo (como geralmente ocorre em sistemas mais vulneráveis, como o Windows).

Caso você tente alterar ou acessar um arquivo sem permissão, a função *fopen* também irá retornar NULL para o ponteiro.

```
if(arquivo == NULL)
   printf("Nao foi possivel abrir o arquivo!");
```

PS: agradecimento ao leitor *steniovm*, que ajudou na correção do artigo :D

Escrevendo em arquivos - As funções fputc(), fprintf() e fputs()

Agora que já aprendemos as operações básicas com arquivos, como abrir e fechar, podemos finalmente trabalhar com eles em nossos projetos.

Neste tutorial de nossa **apostila de C**, iremos ensinar como escrever em arquivos através das funções *fputc*, *fprintf* e *fputs*.

- ## Arquivos padrões - *Standard stdin, stdout* e *stderr*

Antes de iniciarmos a escrita nos arquivos, se faz necessário explicar alguns detalhes sobre como os arquivos são vistos em programação C.

A troca de informações em um sistema pode se dar de várias maneiras.
Até o momento em nosso curso de C, fizemos isso através da interação teclado-programa-tela, onde o usuário fornece os dados via teclado, o programa processa essa informação, e exibe algo na tela.

Outra maneira de trocar informações é através da impressora ou de um scanner.
Ou seja, podemos receber informações através de um dispositivo externo (como um leitor de códigos em barra), bem como podemos mandar dados para uma impressora.

E nesta seção, focaremos em um tipo específico de troca de informações, que é através de arquivos localizados no disco magnético de seu computador. Podemos tanto pegar informações dele, como escrever algo em seu HD através da linguagem C.

Resumindo: existem várias maneiras de se trocar informações, e sempre está surgindo novas. Por exemplo, as tecnologias WiFi e *Bluetooth.* Visando facilitar, as coisas foram padronizadas como sendo arquivos.

Ou seja, quando fornecemos dados por meio de um teclado, essas informações são passadas em série, por meio de *stream*, como se fosse um arquivo, embora este não exista.
Nosso programa em C vai ler esses dados do teclado COMO SE FOSSE um arquivo.
Isso ocorre porque ao criar um programa em C, estamos automaticamente usando alguns tipos de arquivos.

Mesmo que apenas tenhamos a interação teclado-programa-tela, existem alguns arquivos abertos, e os mais importantes são:

- **stdin** - é o arquivo de entrada padrão. Até o momento, para nós, foi o teclado.
- **stdout** - é o arquivo de saída padrão. Até o momento, nossa saída padrão é a tela do computador, através do terminal de comando.
- **stderr** - arquivo padrão de erro, onde podemos direcionar mensagens de erro para outro local sem ser o da saída padrão, como para um *log* (espécie de registro de ações)

Além desses, outros arquivos como o *stdaux* (dispositivo auxiliar, geralmente a porta COM1) e o *stdprn*(impressora padrão), são abertos automaticamente. Por isso, podemos usar a vontade funções como *printf*, *scanf*, *putc* e outras.

Mas agora não vamos mais usar essas entradas e saídas padrão, e sim arquivos, por isso iremos fornecer algumas informações especiais aos nossos programas, informações sobre as entradas e saídas, que serão em formas de arquivos salvos na sua máquina.

fputc() - Como escrever um caractere em um arquivo

Sem mais delongas, vamos as vias de fato e finalmente aprender a criar e escrever em um arquivo externo de nosso computador.

Para fazer isso vamos usar a função fputc(), que recebe dois dados: o caractere e o *FILE**, que terá as informações do local onde iremos escrever o dito caractere:

```
int fputc(int char, FILE *arq);
```

Essa função retorna *EOF* caso não tenha conseguido escrever no arquivo, ou retorna o inteiro que representa o caractere, caso tenha ocorrido.

- ### Exemplo de código - Escrevendo um caractere em um arquivo

Escreva um programa que peça um caractere para o usuário e salve esta entrada em um arquivo chamado "char.txt", localizado na mesma pasta do programa executável.

Vamos inicialmente definir nosso endereço através de uma string (*char url[]*), que é simplesmente "char.txt", bem como o caractere *ch* para armazenar o caractere que o usuário digitar.

Como queremos salvar o *char* em um arquivo, criamos um ponteiro *arq* do tipo *FILE*.
Em seguida, pedimos para o usuário digitar algum caractere, que capturamos através da função *getchar()* e salvamos em *ch*.

Agora vamos abrir um arquivo para escrever, e isso é feito através da função *fopen()*, que vai receber a string com a localização do arquivo (isto está armazenado na variável *url*) e o modo de abertura.
Como queremos escrever, o modo é o *"w"*.

Para checarmos se abertura do arquivo foi realmente efetuada, testamos se a *fopen* não retornou NULL para *arq*. Caso não, vamos escrever o caractere no arquivo.
Isso é feito pela função *fputc*, que diferente da *putchar* que necessita só do caractere, ela também necessita saber onda vai ser a saída (pois não é mais a saída padrão, a tela).

Essa saída é indicada pelo vetor *arq*. E pronto, caractere salvo no arquivo "char.txt".
Mas antes de finalizar o programa, devemos adotar a boa prática de programação que nossa **apostila de C** ensinou: fechar os arquivos, usando a *fclose* e passando o *arq* como argumento.
Nosso código C ficou:

```c
#include <stdio.h>

int main(void)
{
	char url[]="char.txt";
	char ch;
	FILE *arq;

	printf("Caractere: ");
	ch=getchar();
```

```
        arq = fopen(url, "w");
        if(arq == NULL)
                printf("Erro, nao foi possivel abrir o arquivo\n");
        else{
                fputc(ch, arq);
                fclose(arq);
        }

        return 0;
}
```

Pronto, agora vá na pasta onde está localizado seu executável e vai ver que foi criado um arquivo chamado "*char.txt*", e dentro dele está o caractere que você digitou.

Pode fechar o programa, reiniciar a máquina ou fazer o que quiser, que o dito cujo arquivo vai continuar no seu HD.

Bacana, não?
Agora experimente rodar de novo o programa, escrevendo outro caractere. Veja que o antigo foi substituído. Isso ocorreu devido ao modo de abertura, "w".

- **Exemplo de código - Adicionando caractere com o modo "a" (*append*)**

Crie um programa semelhante ao anterior, mas em vez de substituir o caractere, ADICIONE um caractere ao fim do arquivo.
Escreva no arquivo: "C Progressivo"

Para escrever um texto em sequência, temos que alterar apenas uma coisa: o modo de abertura, de "w" para "a", assim sempre que escrevermos algo, será inserido ao final, anexando, em vez de substituir.

Vamos criar um looping do while para pedir caracteres ao usuário, e este laço só termina se digitarmos enter (caractere '\n').
Como vamos 'mexer' várias vezes no arquivo, abrimos ele antes do looping e fechamos depois.

Nosso código fica:

```c
#include <stdio.h>

int main(void)
{
	char url[]="char.txt";
	char ch;
	FILE *arq;

	arq = fopen(url, "a");
	if(arq == NULL)
		printf("Erro, nao foi possivel abrir o arquivo\n");
	else
	do{
		printf("Caractere: ");
		ch=getchar();
		fflush(stdin);

		fputc(ch, arq);
	}while(ch != '\n');

	fclose(arq);

	return 0;
}
```

Digite caractere por caractere, inclusive espaço, o seguinte: C Progressivo
Em seguida dê enter para terminar.
Veja como ficou seu arquivo *char.txt*

Agora rode o programa de novo, e digite outra coisa qualquer e dê enter ao final.
Veja novamente seu arquivo! Vai ver que o que tinha antes não foi apagado, e o que digitou depois está na linha de baixo.
Interessante, não?

- ***fprintf()*** **- Escrevendo textos (strings) em arquivos**

Embora interessante, e importante, a escritura de caracteres em arquivos, ela é um pouco limitada.

Afinal, é incômodo escrever caractere por caractere.

Para isso, existe a função *fprintf*, que nos permite escrever strings inteiras em arquivos.
Sua sintaxe simplificada é:
int fprintf(FILE *arq, char string[])

Ou seja, recebe o local onde deve direcionar a saída (para um arquivo, apontado pelo ponteiro *arq* do tipo *FILE*), e a string que devemos adicionar em tal arquivo.
Essa função retorna *EOF* em caso de erro.

- **Exemplo de código - Armazenando notas, calculando a média e escrevendo no arquivo**

Escreva um programa em C que peça 3 notas de um aluno (Matemática, Física e Química), e salve esses dados em um arquivo chamado "notas.txt", que deve ter, ao final, a média das três disciplinas.

Vamos definir a nossa url como "notas.txt" e criar duas variáveis do tipo *float*, a "nota" (que vai armazenar a nota de cada matéria) e a "media" (que vai somar todas as notas, depois se dividir por 3 para ter a média).

A cada vez que pedimos uma nota, adicionamos uma linha de informação ao nosso arquivo, e ao final escrevemos a média.
O interessante é que nossa saída de dados é o arquivo, e não mais a tela. Assim, embora exista saída, nós não vemos na tela, só no arquivo.

Veja como ficou nosso código:

```
#include <stdio.h>

int main(void)
{
	char url[]="notas.txt";
	float nota,
	    media=0.0;
```

```
	FILE *arq;

	arq = fopen(url, "w");
	if(arq == NULL)
		printf("Erro, nao foi possivel abrir o arquivo\n");
	else{
		printf("Nota de Matematica: ");
		scanf("%f", &nota);
		fprintf(arq, "Matematica: %.2f\n", nota);
		media+=nota;

		printf("Nota de Fisica: ");
		scanf("%f", &nota);
		fprintf(arq, "Fisica: %.2f\n", nota);
		media+=nota;

		printf("Nota de Quimica: ");
		scanf("%f", &nota);
		fprintf(arq, "Quimica: %.2f\n", nota);
		media+=nota;

		media /= 3;
		fprintf(arq, "Media final: %.2f\n", media);
	}
	fclose(arq);

	return 0;
}
```

E se a *fputc* é a equivalente da *putc*, bem como a *fprintf* é a equivalente da *printf*, fica óbvio saber qual a função da *fputs*: é a mesma da *puts*, cuja a saída é uma string com o *new line* "\n" no final.

Exercício sobre escrita de dados em C

Escreva um programa que peça números decimais ao usuário, até que ele digita 0.
Salve todos os números digitados em um arquivo chamado "dados.txt", que mostra a média desses números no final do arquivo.

Lendo arquivos em C: As funções fgetc, fscanf e fgets

Agora que já aprendemos a escrever em arquivos em C, vamos aprender agora em nossa **apostila de C** a outra parte: aprender como ler informações dos arquivos!

Iremos aprender como ler caractere por caractere, ou um trecho pré-formatado ou linha por linha.
Isso vai depender da necessidade de cada aplicativo em C.

- ***fgetc()*** **- Como ler caracteres de um arquivo**

Se você leu o tutorial passado de nosso curso de C, vai notar que não tem muita diferença nem dificuldade na leitura de informações de um arquivo.

Quando estávamos trabalhando com os arquivos padrões (lendo informações do teclado), usávamos funções como *getchar*, *scanf* e *gets*. Aqui, iremos ensinar a usar funções correspondentes, porém para trabalhar com arquivos que estão armazenados em nosso disco rígido.

A única diferença entre leitura e escrita, é a posição do arquivos que vamos ler.
Quando estávamos escrevendo, se usássemos o modo de abertura "w", escrevíamos desde o início do arquivo.
Se fosse usando o "a", escrevíamos ao final do arquivo.

Já na leitura, vamos usar funções que iniciarão a leitura sempre do começo do arquivo.
Quando usamos a função *fgetc*, por exemplo, ele lê o primeiro caractere e automaticamente já se posiciona no próximo. Ao ler o próximo, o C já se prepara para ler o próximo caractere do arquivo, e assim segue, até encontrar a constante ***EOF***.

A sintaxe da função *fgetc* é:
```
int fgetc(FILE *arq)
```

Ela retorna um inteiro que representa o caractere, e ***EOF***, que vale -1, caso aponte para o fim do arquivo. E como os caracteres são representados por inteiros que vão de 0 até 255, um caractere no arquivo nunca terá o valor -1, somente entre 0 e 255.

E como comentamos, uma coisa importante que ocorre 'por debaixo dos panos' é que após retornar um caractere, está função já passa a apontar para o próximo caractere, automaticamente, até encontrar -1 (***EOF***).

- **Exemplo de código C - Programa que lê o conteúdo de um arquivo, caractere por caractere**

Faça um programa que leia de um arquivo "poema.txt" e imprima na tela, caractere por caractere.

Primeiramente, escreva algo no "poema.txt", pode ser um poema, seu nome completo, endereço etc.
Escreva algo e coloque esse arquivo na mesma pasta do executável.
Assim, nosso endereço é simplesmente "poema.txt", e como queremos ler, vamos usar o método de abertura "r".

Declaramos uma variável de nome "ch" do tipo *char*, que vai receber caractere por caractere do arquivo.
Isso é feito usando o seguinte código:
ch=fgetc(arq)

E isso deve ser feito enquanto o caractere apontando no arquivo não for ***EOF***.
Fazemos essa checagem assim:
(ch=fgetc(arq))!= EOF

Assim, nosso programa que mostra todo o conteúdo de um arquivo na tela, é:

```
#include <stdio.h>

int main(void)
{
	char url[]="poema.txt";
	char ch;
	FILE *arq;

	arq = fopen(url, "r");
	if(arq == NULL)
	   printf("Erro, nao foi possivel abrir o arquivo\n");
```

```
        else
            while( (ch=fgetc(arq))!= EOF )
                putchar(ch);

        fclose(arq);

        return 0;
}
```

- **Exercício resolvido - Contando o número de linhas de um arquivo**

Escreva um programa que conte o número de linha do arquivo da questão anterior.

O que caracteriza uma linha?
O caractere *new line* "\n".

Para contar quantas linhas tem um arquivo de texto, basta percorrermos todos os caracteres do arquivo em busca dos "\n", pois cada caractere desse geralmente está no final de uma linha.

Vamos usar o código do exemplo passado deste tutorial para percorrer todos os caracteres, e cada vez que encontrar um "\n" incrementamos a variável "num", inicializada com 0.

Nosso código é:

```
#include <stdio.h>

int main(void)
{
        char url[]="poema.txt",
             ch;
        int num=0;
        FILE *arq;

        arq = fopen(url, "r");
        if(arq == NULL)
                        printf("Erro, nao foi possivel abrir o arquivo\n");
        else
                while( (ch=fgetc(arq))!= EOF )
```

```
            if(ch == '\n')
              num++;

    printf("Existem %d linhas no arquivo\n", num);
    fclose(arq);

    return 0;
}
```

- ***fscanf()*** **- Lendo uma entrada formatada**

Assim como é possível escrever de maneira formatada através das funções *printf* e *fprintf*, também podemos ler de maneira formatada, como é possível na *scanf*, através do uso da função *fscanf*.

Isso é particularmente interessante se tivermos um arquivo com um determinado formato.
Ou seja, quando o conteúdo do arquivo obedecer um determinado padrão.

Por exemplo, vamos supor que você tenha uma lista com notas de 3 alunos. Na primeira coluna as notas de Matemática, na segunda as de Física e na terceira coluna as notas de Química.

Cada linha é representada como: "%f %f %f\n"
Ou seja: número, espaço em branco, número, espaço em branco, número.e o *new line*.
Este é o formato, este é o padrão.

A sintaxe da *fscanf* é:
int fscanf(FILE *arq, char *string_formatada)

Como as outras, retorna ***EOF*** caso não tenha conseguido fazer a leitura de maneria correta.

- **Exemplo de código - Como usar a *fscanf***

Suponha que tenhamos um arquivo de texto chamado "arquivo.txt" com o seguinte conteúdo:
a b c
d e f
g h i

j k l
Como usar a *fscanf* para lê-lo?

Basta notar que o formato desse arquivo é: caractere, espaço, caractere, espaço, caractere e enter
Ou seja: "%c %c %c\n"
O formato se repete lina por linha, onde temos que receber 3 caracteres por linha.

Então vamos salvar esses três em três variáveis do tipo *char* e exibi-las.

Nosso código para ler e exibir esses caracteres é:

```
#include <stdio.h>

int main(void)
{
	char url[]="arquivo.txt",
	    ch1, ch2, ch3;
	FILE *arq;

	arq = fopen(url, "r");
	if(arq == NULL)
			printf("Erro, nao foi possivel abrir o arquivo\n");
	else
		while( (fscanf(arq,"%c %c %c\n", &ch1, &ch2, &ch3))!=EOF )
			printf("%c %c %c\n", ch1, ch2, ch3);

	fclose(arq);

	return 0;
}
```

- **Exemplo de código - Nomes, notas e média**

Em um arquivo chamado "notas.txt" está os dados dos nomes e notas de alunos.
Em cada linha há o nome do aluno, seguido de três notas:

Maria 8 8 10
Jose 6 6 8
Carlos 7 9.5 7.5

Programador 10 10 10

Crie um programa que exiba o nome de cada aluno e sua média.

A primeira coisa que devemos fazer é analisar o conteúdo do arquivo e procurar por padrões.
Nesse caso, o padrão se repete em toda linha, pois todas as linhas são iguais: string, espaço, número, espaço, número, espaço, número, enter
Ou seja: "%s %f %f %f\n"

Como há um padrão em toda linha, podemos usar a *fscanf* para armazenar esses dados.
Vamos precisar de uma string e três *floats*.
Após pegar esses dados, exibimos o nome do aluno e a média das notas.

Nosso código em C fica:

```
#include <stdio.h>

int main(void)
{
	char url[]="notas.txt",
	     nome[20];
	float nota1, nota2, nota3;
	FILE *arq;

	arq = fopen(url, "r");
	if(arq == NULL)
			printf("Erro, nao foi possivel abrir o arquivo\n");
	else
		while( (fscanf(arq,"%s %f %f %f\n", nome, &nota1, &nota2,
&nota3))!=EOF )
			printf("%s teve media %.2f\n", nome, (nota1+nota2+nota3)/3);

	fclose(arq);

	return 0;
}
```

- ***fgets()*** **- Capturando linha**

Usávamos a função *gets* para capturar uma string do teclado do usuário.

Ela pegava do primeiro caractere até encontrar o *new line*("\n"), e colocava o caractere delimitador ao final ("\0").

Muitas vezes é interessante pegar uma linha inteira de um arquivo, principalmente se nesse arquivo existir textos.
Para fazer esta tarefa, uma boa opção é usar a função *fgets*, cuja sintaxe é:
char *fgets(char *minhaString, int numBytes, FILE *arq)

Essa função vai abrir o arquivo apontado por *arq* e vai pegar do primeiro caractere até o *new line*, ou até o limite de "numBytes" *bytes* e vai armazenar essa string na "minhaString".

Ou seja, vamos usar ela para pegar cada linha de um arquivo e armazenar na forma de string.
Por exemplo, escreva em um arquivo "dados.txt"o seguinte conteúdo:

"Meu nome: [escreva seu nome completo]
Moro em: [escreva seu endereço]
Estudo pelo C Progressivo
E pretendo ser programador C"

Agora vamos criar um programa que vai ler e exibir esses dados:

```
#include <stdio.h>

int main(void)
{
	char url[]="dados.txt",
	     info[50];
	FILE *arq;

	arq = fopen(url, "r");
	if(arq == NULL)
		printf("Erro, nao foi possivel abrir o arquivo\n");
	else
		while( (fgets(info, sizeof(info), arq))!=NULL )
			printf("%s", info);

	fclose(arq);

	return 0;
```

```
}
```

Podemos usar cada linha dessas e armazenar em uma string diferente (string que guarda o nome, outra que guarda o endereço, outra que guarda o CPF, RG etc, igual aqueles formulários que preenchemos na internet).

Vale notar que, como a *fgets* retorna uma string, para checar se chegamos ao fim do arquivo, basta checarmos se o retorno dela é diferente de *NULL*.