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.

Baixe o conteúdo do site: [Apostila de C](https://www.cprogressivo.net/p/apostila-c-progressivo.html)

[Obtenha sua certificação de programação C para entrar no Mercado de Trabalho!](http://afiliado.bravacursos.com.br/4573/curso-de-programacao-em-c/" \t "_blank)

Tipos de Listas em C

No esquema que mostramos no tutorial passado de nossa [apostila de C](http://www.cprogressivo.net/), explicamos [como funciona uma lista](http://www.cprogressivo.net/2013/10/Listas-em-C-O-que-e-como-funciona-uma-LIST.html), 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 encadeada 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](http://www.cprogressivo.net/2013/02/O-teste-condicional-SWITCH-o-que-e-para-que-serve-e-como-usar-o-switch-em-C.html).

|  |  |
| --- | --- |
|  |  |

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()](http://www.cprogressivo.net/2013/04/Como-usar-a-funcao-malloc-para-alocar-memoria-em-linguagem-C.html), conforme estudamos na seção de nosso curso de [Alocação Dinâmica de Memória](http://www.cprogressivo.net/p/alocacao-dinamica-de-memoria-em-c.html):  
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](http://www.cprogressivo.net/2013/02/O-que-e-e-como-usar-o-laco-WHILE-em-C.html), 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](http://www.cprogressivo.net/2013/10/Funcao-free-Como-liberar-memoria-e-evitar-vazamento.html), 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

#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);

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));

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.

-------------------------------------------------------------------------

-------------------------------------------------------------------------

-------------------------------------------------------------------------

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.

Leia este tutorial no seu computador ou celular: Apostila C Progressivo

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.

Baixe o conteúdo do site: Apostila de C

Quer se tornar um programador profissional?

Clique aqui e obtenha seu certificado.

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.

-------------------------------------------------------------------------

-------------------------------------------------------------------------

-------------------------------------------------------------------------

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().

Download do curso: Apostila C Progressivo

Entre no mercado de trabalho! Clique aqui e obtenha seu certificado de programação C!

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:

#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:

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]);

}

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](http://www.cprogressivo.net/p/curso-de-c-online-para-iniciantes.html), 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](http://www.cprogressivo.net/2013/03/O-que-e-alocacao-dinamica-de-memoria-em-Linguagem-C.html), junto com os artigos sobre a [função malloc()](http://www.cprogressivo.net/2013/04/Como-usar-a-funcao-malloc-para-alocar-memoria-em-linguagem-C.html) e sobre a [liberação de memória com a função free()](http://www.cprogressivo.net/2013/10/Funcao-free-Como-liberar-memoria-e-evitar-vazamento.html).

Leia este tutorial no seu computador ou celular: [Apostila C Progressivo](https://www.cprogressivo.net/p/apostila-c-progressivo.html)

[Clique aqui e obtenha seu certificado de programação C e entre no Mercado de Trabalho !](http://afiliado.bravacursos.com.br/4573/curso-de-programacao-em-c/)

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](http://www.cprogressivo.net/2013/01/O-testecondicional-IF-ELSE.html) 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(1000000000000000000);

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](http://www.cprogressivo.net/2013/02/O-teste-condicional-SWITCH-o-que-e-para-que-serve-e-como-usar-o-switch-em-C.html) que irá tratar cada opção descrita pelo usuário.  
Esse switch está dentro de um [laço do while](http://www.cprogressivo.net/2013/02/O-que-e-e-como-usar-o-laco-DO-WHILE-em-linguagem-C.html), 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](http://www.cprogressivo.net/p/aprenda-tudo-sobre-ponteiros-em-c.html) 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>