Organização e Recuperação da Informação

Programação em C

Ponteiros

Leard de Oliveira Fernandes
CET 082

Programação em C

- Ponteiros
  - O sólido entendimento de ponteiros, além da habilidade de utilizá-los efetivamente é de fundamental importância para programadores C;
- A sua definição é simples
  - Variável que armazena o endereço de uma posição da memória.
- A chave para entendimento de ponteiros é entender como a memória é gerenciada num programa em C

Ponteiros e Memória

- Quando um programa em C é compilado, ele trabalha com três tipos de memória:
  - Global/Estática
  - Automática
  - Dinâmica

Ponteiros e Memória

Global/Estática

- Variáveis declaradas estaticamente são alocadas nesta região da memória (static);
- Variáveis globais também utilizam esta região da memória;
- Elas são alocadas quando o programa é iniciado, e sua existência permanece até que o programa termine;
- Enquanto todas as funções possuem acesso as variáveis globais, o escopo das variáveis estáticas é restrita à sua função de definição;

Ponteiros e Memória

Automática (Local)

- Variáveis declaradas dentro de uma função e são criadas quando uma função é executada;
- O seu escopo é restrito à função, e o seu tempo de vida é determinado pelo tempo de funcionamento da função;

Ponteiros e Memória

Dinâmica

- A memória é alocada do *Heap* e pode ser liberada quando necessário;
- Um ponteiro referencia a memória alocada
- O escopo é limitado ao ponteiro ou ponteiros que referenciam a memória

Ponteiros e Memória

Tipo	Escopo	Duração (lifetime)
Global	Arquivo inteiro	O tempo da aplicação
Estático	Função em que é declarada	O tempo do aplicação
Local	Função em que é declarada	Enquanto a função está sendo executada
Dinâmico	Determinada pelo ponteiro que referencia a memória	Até que a memória seja liberada

Ponteiros e Memória

- Uma variável ponteiro contém o endereço de memória de outra:
  - Variável,
  - Objeto, ou
  - Função

Ponteiros e Memória

- Um objeto é considerado alocado na memória quando se utiliza uma das funções de alocação de memória
  - Função malloc
- Um ponteiro é normalmente declarado possuindo um tipo específico
  - char, int, ...
- Um objeto pode ser qualquer tipo de dado do C
  - Inteiro, caracter, string ou estrutura
- Entretanto, nada dentro de um ponteiro indica o tipo de dados que um ponteiro está referenciando

Ponteiros e Memória

- Lembre-se um ponteiro contém apenas um endereço

Ponteiros

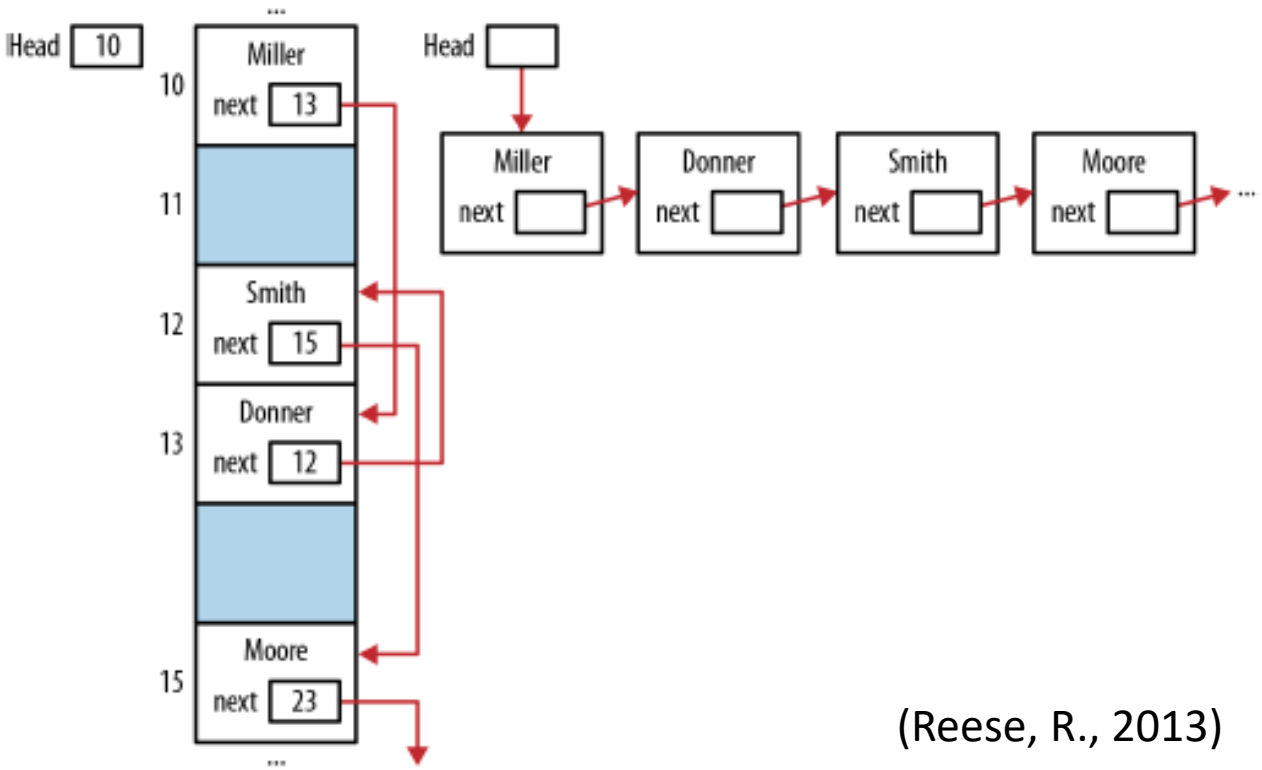
- Ponteiros possuem diversas utilidades
  - Torna a criação de código mais rápida e eficiente
  - Fornece um meio conveniente para endereçar muitos tipos de problemas
  - Suporta alocação dinâmica de memória
  - Torna expressões compactas e sucintas
  - Fornece a habilidade de passar estruturas de dados por apontamento sem criar grande sobrecarga (overhead)
  - Protege dados passados como parâmetro para uma função

Ponteiros

- Ponteiros possuem diversas utilidades
  - Torna a criação de código mais rápida e eficiente
  - Fornece um meio conveniente para endereçar muitos tipos de problemas
  - Suporta alocação dinâmica de memória
  - Torna expressões compactas e sucintas
  - Fornece a habilidade de passar estruturas de dados por apontamento sem criar grande sobrecarga (overhead)
  - Proteger dados passados como parâmetro para uma função

Ponteiros

Uma lista ligada pode ser implementada usando ponteiros ou arrays



(Reese, R., 2013)

Ponteiros

Uma lista ligada pode ser implementada usando ponteiros ou arrays

- Arrays
  - Inserir mais itens do que o limite do array?
  - Inserir um elemento no meio do array?
- Ponteiros
  - “Não há limitação de tamanho”
  - Inserir um elemento no centro da lista é trivial, basta mudar os ponteiros

Ponteiros

- Mas, com grandes poderes há grandes responsabilidades... (uncle ben)
  - Acessar arrays e outras estruturas de dados fora de seus limites;
  - Referenciar variáveis locais após elas deixarem de existir;
  - Referenciar a memória heap alocada após ela ter sido liberada;
  - “Dereferenciar” um ponteiro antes da memória ser alocada à ele;

Ponteiros

- Apesar da sintaxe e semântica de utilização dos ponteiros estar descrita na implementação do C, o comportamento de ponteiros em algumas situações não são definidas
  - Dependente de implementação
  - Não especificada
    - Não há documentação sobre a implementação
  - Indefinidas
    - Não existe requerimento imposto, e qualquer coisa pode acontecer
- Podem existir comportamentos locais;

Ponteiros

Declarando ponteiros

- Tipo de dados
- Asterisco
- Nome da variável ponteiro

```
int num;
int *ptr;
```

- A utilização de espaços em torno do asterisco é irrelevante

```
int num;
int* ptr;
int * ptr2;
int *ptr3;
int*ptr4;
```

Ponteiros

- O asterisco declara uma variável como ponteiro
  - Sobrecarga do símbolo *
    - Multiplicação
    - “Dereferenciamento”

Ponteiros

```
int num;
int *ptr;
```

num (100)	...
ptr (104)	...
108	...

- Ao lado temos o estado da memória após a declaração das variáveis no código anterior
  - Supondo que num esteja alocada na posição 100 e ptr na posição 104;
  - E que ambos ocupem 4 Bytes;
- Ponteiros não inicializados podem ser um problema. Se um ponteiro for “dereferenciado”, o conteúdo do ponteiro não representará um endereço válido;
  - Se isso ocorrer ele pode não conter dados válidos
- Um endereço inválido, é aquele que um programa não está autorizado a acessar;
  - O que em algumas plataformas pode significar o fim da execução do programa

Ponteiros

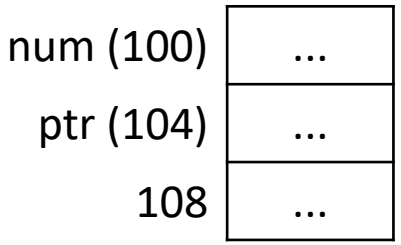
```
int num;
int *ptr;
```

num (100)	...
ptr (104)	...
108	...

- Lembre-se
  - O conteúdo de ptr deverá, eventualmente, ser associado ao endereço de uma variável inteira;
  - Estas variáveis não foram inicializadas, ou seja, talvez possuam lixo;
  - Não há nada inerente à implementação de ponteiros que determine qual o tipo de dados que ele está referenciando ou que seu conteúdo seja válido;
  - Entretanto, o tipo de ponteiro é especificado e o compilador saberá quando o tipo usado não é o correto;

Ponteiros

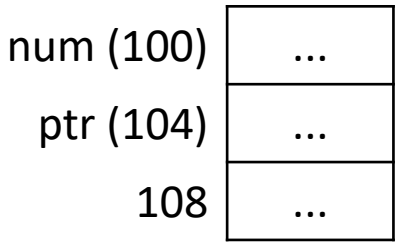
```
int num;
int *ptr;
```



- Como ler uma declaração de ponteiros
  - 1 - ptr é uma variável;
  - 2 – ptr é uma variável ponteiro;
  - 3 - ptr é uma variável ponteiro para um inteiro;

Ponteiros

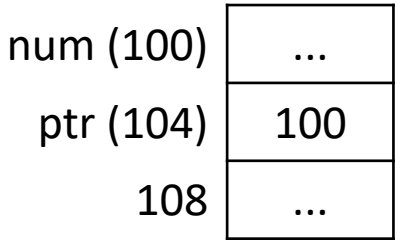
```
int num;
int *ptr;
```



Operador de Endereçamento (&)

- Retorna o endereço de seu operando;
- Podemos inicializar ptr

```
int num;
int *ptr=&num;
```



- Usar esta declaração leva a um erro ou aviso

```
int num;
int *ptr=num;
```

```
int num;
int *ptr;
ptr = num;
```

Ponteiros

```
int num=0;
int *ptr=&num;
```

num (100)	0
ptr (104)	100
108	...

Imprimindo valores dos ponteiros

- Os endereços das variáveis podem ser determinados, imprimindo sua saída.

```
int num=0;
int *ptr=&num;

printf("Endereço de num: %d, Valor de num: %d, &num, num);
printf("Endereço de ptr: %d, Valor de ptr: %d, &ptr, ptr);
```

Ponteiros

```
int num=0;
int *ptr=&num;
```

num (100)	0
ptr (104)	100
108	...

- Semelhanças a parte (filme matrix), os endereços que estão no ponteiro não são reais...
  - São virtuais (Assunto abordado em Sistemas Operacionais)
- Para o programa pouco importa, ele acredita que está acessando os endereços reais e não sabe da presença do S.O.
  - Para o programador também!!! (Ố ṻ Ồ)

Ponteiros

```
int num=0;
int *ptr=&num;
```

num (100)	0
ptr (104)	100
108	...

Dereferenciando um ponteiro

- O operador de indireção, *, retorna um valor apontado por uma variável ponteiro;
  - Isto é referenciado como “dereferenciando um ponteiro”
- Este código irá imprimir o conteúdo de num

```
printf(“%d\n”, *ptr);
```

- Podemos utilizar o resultado do dereferenciamento com um lvalue;
  - Operando que está no lado esquerdo do operador de atribuição
  - Lvalues devem ser modificáveis desde estejam sendo associados a um valor

```
*ptr = 666;
printf(“%d\n”, num);
```

num (100)	666
ptr (104)	100
108	...

Ponteiros

NULL

- Quando um NULL é associado a um ponteiro, significa que este ponteiro aponta para nada;
- A ideia é que um ponteiro possa armazenar um valor que não é igual a outro ponteiro;
- Ele não aponta para qualquer região da memória;
- Dois ponteiros nulos serão sempre iguais um ao outro;
- O conceito de nulo é uma abstração suportada por um ponteiro constante nulo;
- Ponteiros nulos não podem ser dereferenciados;

Ponteiros

NULL

- A macro NULL é um zero inteiro constante para um ponteiro para void

```
#define NULL ((void*)0)
```

- Como já podem ter observado, o conceito de ponteiro nulo é importante para muitas implementações de estruturas de dados;

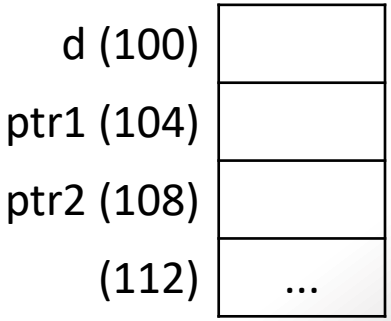
```
ptr = NULL;
```

Ponteiros

- Estas instruções estão corretas?

```
01 pi = 0;
02 pi = NULL;
03 pi = 100;
04 pi = num;
```

```
int d = 40;
int *ptr1=0, ptr2=0;
ptr1 = d;
ptr2 = d;
printf("%d\n", ptr2);
printf("%d\n", ptr1);
```



Ponteiros

- Um ponteiro pode ser utilizado como operando único de uma expressão lógica

```
if(ptr){
   //Não Nulo
} else {
   //Nulo
}
```

Ponteiros

Ponteiro para void

- Um ponteiro para o tipo void é um ponteiro de propósito geral utilizado para referenciar qualquer tipo de dado;

```
void *ptrv;
```

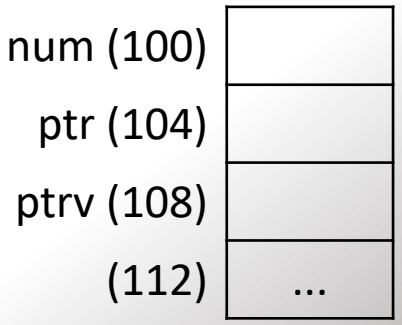
- Um ponteiro para void terá a mesma representação e alinhamento de memória como um ponteiro para char;
- Um ponteiro para void nunca será igual a outro ponteiro. Entretanto dois ponteiros void referenciados por um valor NULL serão iguais.

Ponteiros

Ponteiro para void

- Qualquer ponteiro pode ser atribuído para um ponteiro void;
- Você pode utilizar o **cast** para voltar ao ponteio original;
  - Quando isso ocorre, o valor será igual ao valor do ponteiro original

```
int num;
int *ptr = &num;
printf("Valor de ptr: %d\n", ptr);
void* ptrv = ptr;
ptr = (int*) ptrv;
printf("Valor de ptr: %d\n", ptr);
```



Ponteiros

Ponteiro para void

- O operador sizeof pode ser utilizado com um ponteiro void;
  - Não pode ser utilizado com o tipo void;

```
size_t size = sizeof(void*); //Correto
size_t size = sizeof(void); //incorreto
```

Ponteiros

Operador sizeof

- O operador sizeof pode ser utilizado para determinar o tamanho de um ponteiro

```
printf("Tamanho de um char*: %d\n", sizeof(char*));
```

- O operador sizeof retorna o tamanho do ponteiro char.
  - Este valor é definido como do tipo size_t é uma variável do tipo inteiro sem sinal;
    - Typedef unsigned int

```
#ifndef __SIZE_T
#define __SIZE_T
typedef unsigned int size_t;
#endif
```

Ponteiros

Operadores de Ponteiros

Operador	Nome	Descrição
*		Usado para declarar um ponteiro
*	Dereferenciar	Usado para dereferenciar um ponteiro
->	Aponta-para	Usado para acessar campos de uma estrutura referenciada por ponteiro
+	Soma	Soma dois ponteiros
-	Subtração	Subtrai dois ponteiros
== , !=	Igualdade, Inigualdade	Compara dois ponteiros
> , >= , < , <=	Maior que, Maior ou igual que, Menor que, Menor ou igual que	Compara dois ponteiros
(tipo de dado)	Cast	Muda o tipo de um ponteiro

Ponteiros

Aritmética de Ponteiros

- As operações aritméticas sobre ponteiros incluem:
  - Somar um inteiro a um ponteiro;
  - Subtrair um inteiro de um ponteiro;
  - Subtrair dois ponteiros um do outro;
  - Comparar ponteiros;

Ponteiros

Somar um inteiro a um ponteiro

- Operação comum e bastante útil;
- Quando somamos um inteiro a um ponteiro, o total somado é o **produto** do **inteiro** pelo **número de bytes** do tipo de dados;

Tipo de dados	Tamanho (Bytes)
Byte	1
Char	1
Short	2
Int	4
Long	8
Float	4
double	8

Ponteiros

Somar um inteiro a um ponteiro

- Para observar os efeitos da soma de um inteiro ao ponteiro, usaremos um array de inteiros

```
int vector[] = {36, 61, 72};
int *ptr = vector; //ptr: 100
Printf("%d\n",*ptr); // Mostra 36
ptr += 1; // ptr: 104
printf("%d\n",*ptr); // Mostra 61
ptr += 1; // ptr:108
printf("%d\n",*ptr); // Mostra 72
```

Vector[0] (100)	36
Vector[1] (104)	61
Vector[2] (108)	72
ptr (112)	100

- Se fizermos a operação abaixo, qual o valor de ptr

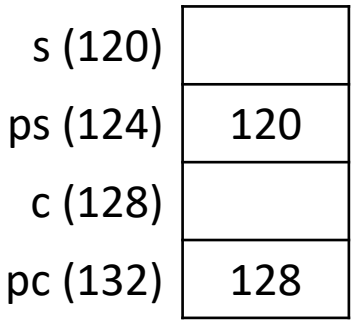
```
int vector[] = {36, 61, 72};
int *ptr = vector;
ptr += 3;
```

Ponteiros

Somar um inteiro a um ponteiro

- Avalie o seguinte código

```
short s;
short *ps = &s;
char c;
char *pc = &c;
```



- Executando as operações abaixo, quais valores são impressos?

```
printf(“Conteúdo de ps antes: %d\n",ps);
ps = ps + 1;
printf("Conteúdo de ps depois: %d\n",ps);
printf("Conteúdo pc antes: %d\n",pc);
pc = pc + 1;
printf("Conteúdo de pc depois: %d\n",pc);
```

Ponteiros

Subtraindo um inteiro de um ponteiro

- Valores inteiros podem ser subtraídos de ponteiros, assim como são somados;

```
int vector[] = {36, 61, 72};
int *ptr = vector+2; //ptr: 108
Printf("%d\n",*ptr); // Mostra 72
ptr--; // ptr: 104
printf("%d\n",*ptr); // Mostra 61
ptr--; // ptr:100
printf("%d\n",*ptr); // Mostra 36
```

Vector[0] (100)	36
Vector[1] (104)	61
Vector[2] (108)	72
ptr (112)	108

Ponteiros

Subtraindo dois ponteiros

- Quando um ponteiro é subtraído do outro, teremos a diferença entre seus endereços em termos de unidade;

```
int vector[] = {36, 61, 72};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;
printf("p2-p0: %d\n",p2-p0); // p2-p0: 2
printf("p2-p1: %d\n",p2-p1); // p2-p1: 1
printf("p0-p1: %d\n",p0-p1); // p0-p1: -1
```

Vector[0] (100)	36
Vector[1] (104)	61
Vector[2] (108)	72
P0 (112)	100
P1(116)	104
P2(120)	108

Ponteiros

Utilização comum de ponteiros

- Ponteiros podem ser utilizados de diversas formas, iremos analisar
  - Múltiplos níveis de indireção;
  - Ponteiros Constantes.

Ponteiros

Múltiplos níveis de indireção

- Ponteiros podem utilizar diferentes níveis de indireção
  - É comum vermos uma variável declarada como ponteiro para um ponteiro, comumente chamada de ponteiro duplo.
  - Um exemplo são os argumentos passados para a função main: argc e argv

Ponteiros

Múltiplos níveis de indireção

- Abaixo temos um exemplo utilizando três arrays
  - O primeiro é um array de strings que armazena uma lista títulos de livros;
  - O segundo e terceiro arrays manterão uma lista de títulos dos melhores livros e dos títulos de origem inglesa

```
char *livros[] = {"A Tale of Two Cities",
"Wuthering Heights", "Don Quixote",
"Odyssey", "Moby-Dick", "Hamlet",
"Gulliver's Travels"};

char **melhores_livros[3];
char **livros_Ingleses[4];
```

Ponteiros

Múltiplos níveis de indireção

- É melhor manter uma cópia do titulo em cada uma das listas ou manter apenas uma referência?
- Quem irá ocupar mais espaço na memória?

```
char *livros[] = {"A Tale of Two Cities",
"Wuthering Heights", "Don Quixote",
"Odyssey", "Moby-Dick", "Hamlet",
"Gulliver's Travels"};

char **melhores_livros[3];
char **livros_Ingleses[4];
```

Ponteiros

Múltiplos níveis de indireção

- Ambos arrays precisam ser declarados como um ponteiro para ponteiro para um char;
- Os elementos do array irão armazenar o endereço dos elementos do array de livros;
- O que evita de termos memória duplicada para cada título de livro;

```
char *livros[] = {"A Tale of Two Cities",
"Wuthering Heights", "Don Quixote",
"Odyssey", "Moby-Dick", "Hamlet",
"Gulliver's Travels"};

char **melhores_livros[3];
char **livros_Ingleses[4];
```

Ponteiros

Múltiplos níveis de indireção

- Abaixo inicializamos os arrays

```
melhores_livros[0] = &livros[0];
melhores_livros[1] = &livros[3];
melhores_livros[2] = &livros[5];

livros_ingleses[0] = &livros[0];
livros_ingleses[1] = &livros[1];
livros_ingleses[2] = &livros[5];
livros_ingleses[3] = &livros[6];
```

Ponteiros

```
melhores_livros[0] = &livros[0];
melhores_livros[1] = &livros[3];
melhores_livros[2] = &livros[5];

livros_ingleses[0] = &livros[0];
livros_ingleses[1] = &livros[1];
livros_ingleses[2] = &livros[5];
livros_ingleses[3] = &livros[6];
```

Múltiplos níveis de indireção

- Abaixo inicializamos os arrays

melhores_livros[0]	100
melhores_livros[1]	112
melhores_livros[2]	120

livros_ingleses[0]	100
livros_ingleses[1]	104
livros_ingleses[2]	120
livros_ingleses[3]	124

livros[0]100	200
livros[1]104	300
livros[2]108	400
livros[3]112	500
livros[4]116	600
livros[5]120	700
livros[6]124	800

200	A Tale of Two Cities
300	Wuthering Heights
400	Don Quixote
500	Odyssey
600	Moby-Dick
700	Hamlet
800	Gulliver's Travels

Ponteiros

Ponteiros Constantes

- Usar o identificador const com ponteiro é algo poderoso no C;
- Este recurso provê diferentes tipos de proteção para diferentes problemas;

Ponteiros

Ponteiros para uma Constante

- Um ponteiro pode ser definido para uma constante;
- Significa que o ponteiro não pode ser utilizado para modificar o valor que ele está referenciando;
- Abaixo temos um inteiro e um inteiro constante declarados. Em seguida um ponteiro para um inteiro e um ponteiro para um inteiro constante

```
int num = 5;
const int numC = 500;
int *pi; // Ponteiro para um int
const int *pci; // Ponteiro para um int constante
pi = &num;
pci = & numC;
```

Ponteiros

Ponteiros para uma Constante

- Não existe problemas no dereferenciamento de um ponteiro de uma constante
  - Desde que seja apenas para leitura
  - Não podemos alterar o valor da constante que ele referencia;
  - Mas nós podemos mudar o seu ponteiro (seja ele constante ou não)
- Esta declaração limita, simplesmente, a nossa capacidade de modificar a variável referenciada pelo ponteiro;

```
printf("%d\n", *pci);

*pci = 300;  //Erro

pci = &num ; //Alterando o ponteiro
```

Ponteiros

```
int num = 5;
const int numC = 500;
int *pi; // Ponteiro para um int
const int *pci; // Ponteiro para um int constante
pi = &num;
pci = & numC;
```

Ponteiro para uma constantes

- A declaração de pci como um ponteiro para uma constante inteira, significa:
  - pci pode ser modificada para apontar para diferentes inteiros constantes;
  - pci pode ser modificada para apontar para diferentes inteiros não constantes;
  - pci pode ser dereferenciado apenas para leitura;
  - pci não pode ser dereferenciado para mudar o que ele está apontando;

```
//Ambas declarações significam a mesma coisa
const int *pci;
int const *pci;
```

Ponteiros

```
int num;
int *const cpi = &num;
```

Ponteiros Constantes para não constantes

- Podemos declarar um ponteiro constante para uma variável não constante;
- Com este tipo de declaração:
  - O ponteiro pode ser inicializado com uma variável não constante;
  - O ponteiro não pode ser modificado;
  - O dado apontado pelo ponteiro pode ser modificado;

```
int num, limite=400;
int *const cpi = &num;

cpi = &limite; //Erro
```

Ponteiros

```
int num, limite=400;
int *const cpi = &num;

*cpi = limite;
*cpi = 40;
```

Ponteiros Constantes para não constantes

- Neste caso, é possível dereferenciar cpi e associá-lo a um novo valor, seja o que o ponteiro estiver referenciando;

```
const int limite=400;
int *const cpi = &limite;

*cpi = 40;
```

- Entretanto, se inicializarmos o ponteiro com uma variável constante, teremos um aviso dado pelo compilador;
  - Ou seja, a variável constante poderá ser modificada
  - O que não é desejável.

Ponteiros

```
int num;
const int limite=400;
const int *const cpci = &limite;
```

Ponteiros Constantes para constantes

- É raramente utilizado, uma vez que o ponteiro não pode ser modificado e nem o valor no qual ele aponta;
- Pode ser utilizado uma variável não constante para ser referenciada
  - Contudo, o seu valor não será modificado por referenciamento;
- O ponteiro deve ser inicializado na declaração


```
int num;
const int limite=400;
const int *const cpci = &num;
```

Ponteiros

Ponteiros Constantes para constantes

- Caso, tente realizar uma das operações abaixo ocorrerá erros

```
int num;
const int limite=400;
const int *const cpci = &limite;

cpci = &num; //Erro

*cpci = 40; //Erro
```

Ponteiros

Ponteiros Constantes para constantes

- Caso, tente realizar uma das operações abaixo ocorrerá erros

```
int num;
const int limite=400;
const int *const cpci = &limite;

cpci = &num; //Erro

*cpci = 40;
```

Ponteiros

Ponteiros para (ponteiros constantes para constantes)

- Ponteiros para constantes também possuem múltiplos níveis de indireção;

```
int num;
const int limite=400;
const int *const cpci = &limite;
const int *const *pcpci = &cpci;
```

Ponteiros

Ponteiros Constantes

Tipo de Ponteiro	Modificação do Ponteiro	Modificação do valor na variável referenciada
Ponteiro para não constante	Sim	Sim
Ponteiro para uma constante	Sim	Não
Ponteiro Constante para não constante	Não	Sim
Ponteiro Constante para uma constante	Não	Não

Gerenciamento da Memória Dinâmica em C

Leard de Oliveira Fernandes
CET 082

Introdução

- Uma das principais aplicações de ponteiros está em utilizar a sua capacidade de referenciar a memória alocada dinamicamente;
- O gerenciamento desta memória através de ponteiros, forma a base para diversas operações
  - Incluindo a manipulação de estruturas de dados complexas
- Para isso, precisamos entender como o gerenciamento da memória dinâmica ocorre no C

Introdução

- A execução de um programa C executa dentro de um sistema de tempo de execução (runtime system)
  - Ambiente implementado por um S.O.
  - Suporta o “stack” e o “heap”...
- O gerenciamento de memória é fundamental para todos os programas
- A habilidade de alocar e desalocar memória permite a uma aplicação sua memória de forma eficiente
  - Ao invés de alocar previamente a memória para acomodar o maior tamanho possível para uma determinada estrutura de dados, somente o tamanho necessário é alocado.

Introdução

- A linguagem C suporta o gerenciamento de memória dinâmica
  - Os objetos são alocados da (na) memória heap;
  - Realizado manualmente, utilizando funções para alocar e desalocar memória;
- Discutiremos sobre alocação e desalocação de memória.
  - Além dos problemas mais comuns ao utilizar ponteiros;

Alocação de Memória Dinâmica

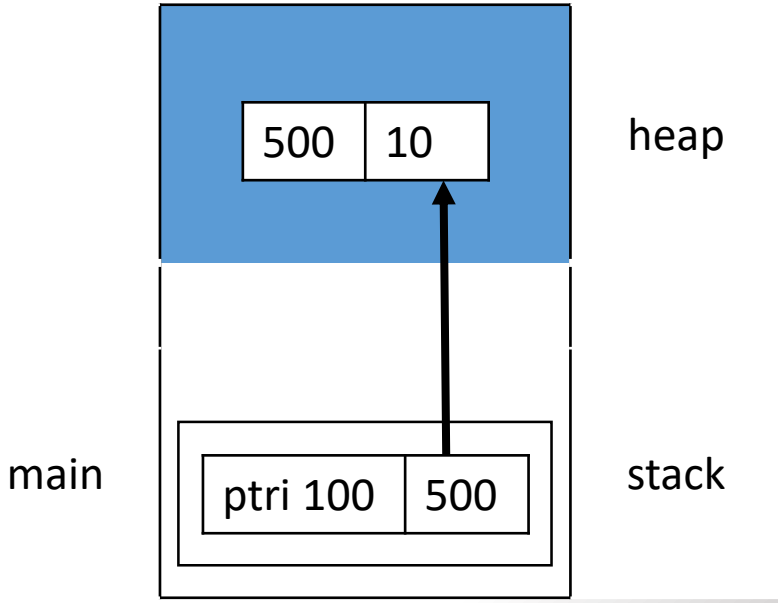
- Os passos básicos utilizados para alocação de memória dinâmica no C, são
  - Usar uma função do tipo **malloc** para alocar memória;
  - Usar esta memória...
  - Desalocar a memória usando a função free
- Existem algumas variações neste procedimento, sendo esta a técnica mais comum.

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
printf("*ptri: %d\n", *ptri);
free(ptri);
```

Alocação de Memória Dinâmica

- Quando o código é executado será impresso o valor 10
  - Caso o mesmo esteja dentro da função main

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
printf("*ptri: %d\n", *ptri);
free(ptri);
```



Alocação de Memória Dinâmica

- O único argumento da função malloc significa o número de bytes a serem alocados;
  - Se tudo ocorrer corretamente, ela retorna um ponteiro para a memória alocada no heap;
  - Caso ocorra uma falha ela retorna um ponteiro NULL;
- O operador sizeof pode ser omitido e você pode inserir diretamente o tamanho de um int (4)

```
int *ptri = (int*) malloc(4);
*ptri = 10;
printf("*ptri: %d\n", *ptri);
free(ptri);
```

Alocação de Memória Dinâmica

- O trecho de código abaixo está correto?

```
int *ptri;
*ptri = (int*) malloc(sizeof(int));
```

Alocação de Memória Dinâmica

- O trecho de código abaixo está correto?
  - Não

```
int *ptri;
*ptri = (int*) malloc(sizeof(int));
```

- Forma correta

```
int *ptri;
ptri = (int*) malloc(sizeof(int));
```

Alocação de Memória Dinâmica

- Lembre-se
  - Cada chamada malloc precisa de uma chamada free correspondente
  - Evita-se o vazamento de memória
  - Uma vez que a memória esteja livre você não deverá acessá-la intencionalmente

Alocação de Memória Dinâmica

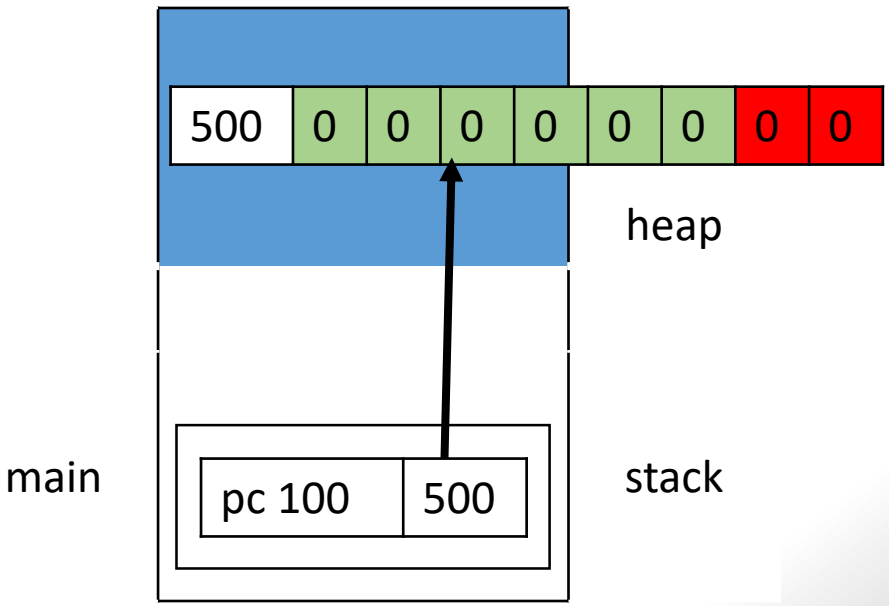
- No geral,
  - Quando a memória é alocada, informações adicionais são armazenadas como parte da estrutura de dados gerenciada pelo gerenciador de heap;
  - Pode conter tamanho do bloco e outras informações;
  - Se a sua aplicação escrever fora deste bloco de memória, a estrutura de dados pode ser corrompida
  - O que leva a comportamentos inesperados;

Alocação de Memória Dinâmica

- Observe o código abaixo

```
char *pc = (char*) malloc(6);
for(int i=0; i<8; i++)
    pc[i] = 0;
```

- Memória extra
  - Depende do compilador
- Teste com valores altos de i > 35



Alocação de Memória Dinâmica

Vazamento de memória

- Ocorre quando uma memória alocada, nunca é utilizada novamente e também não é liberada (free)
- Isto ocorre quando
  - O endereço de memória é perdido
  - A função free nunca é chamada, quando deveria...
- O problema do vazamento é que a memória não pode ser recuperada e utilizada posteriormente
- O total de memória disponível para o gerenciador de heap é diminuida

Alocação de Memória Dinâmica

Vazamento de memória

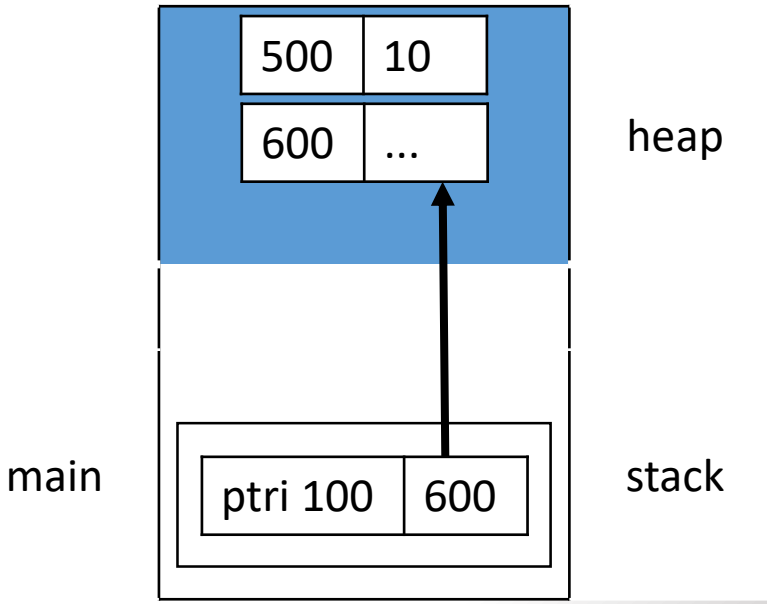
```
char *chunk;
while (1) {
          chunk = (char*)malloc(100*1024*1024);
          printf("Alocando: %d", chunk);
}
```

Alocação de Memória Dinâmica

Perda de Endereço

- Ocorre quando você aloca uma região de memória num segundo momento

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
...
ptri = (int*) malloc(sizeof(int));
```

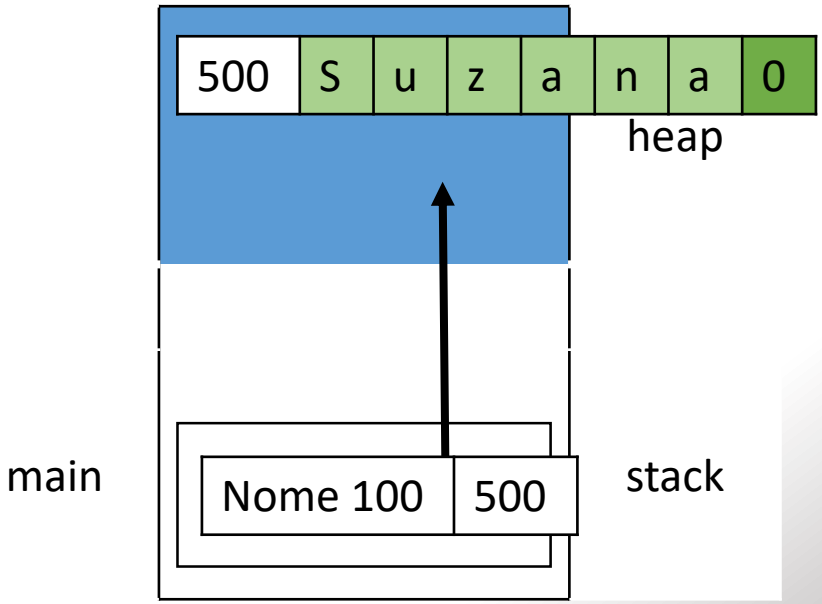


Alocação de Memória Dinâmica

Perda de Endereço

- Ocorre quando você aloca uma região de memória num segundo momento

```
char *nome = (char*)malloc(strlen(“Suzana")+1);
strcpy(nome,"Suzana");
while(*nome != 0) {
        printf("%c",*nome);
        nome++;
}
```



Alocação de Memória Dinâmica

Funções de Alocação Dinâmica

- Existem diversas funções para alocação de memória
- Algumas irão depender da disponibilidade do sistema
- Na biblioteca stdlib.h, temos
  - Malloc
  - Realloc
  - Calloc
  - free

Alocação de Memória Dinâmica

Funções de Alocação Dinâmica

Função	Descrição
malloc	Aloca memória do Heap
realloc	Realoca a memória para um tamanho maior ou menor, baseado num bloco de memória alocado anteriormente
calloc	Aloca e zera a memória do heap
free	Retorna um bloco de memória para o heap

Alocação de Memória Dinâmica

Função malloc

- A função malloc aloca um bloco de memória do heap
- O número de bytes é especificado por um argumento
- O tipo de retorno é um ponteiro para void
- Se não há memória disponível um NULL é retornado
- O bloco de memória retornado pode possuir valores, ou seja, lixo!!!

Alocação de Memória Dinâmica

Função malloc

```
void* malloc(size_t);
```

- Quando a função malloc é executada, ocorrem os seguintes passos
  - Memória é alocada do heap;
  - A memória não é modificada ou limpa;
  - O endereço do primeiro byte é retornado.

```
int *ptri = (int*) malloc(sizeof(int));
```

Alocação de Memória Dinâmica

Função malloc

- Sempre que alocar, faça:

```
int *ptri = (int*) malloc(sizeof(int));
if(ptri != NULL) {
    // Ponteiro válido
} else {
    // Ponteiro Inválido
}
```

Alocação de Memória Dinâmica

Função malloc

- Não esqueça de alocar a memória

```
int *ptri;
printf("%d\n", *ptri);
```

```
char *nome;
printf("Digite um nome: ");
scanf("%s", nome);
```

Alocação de Memória Dinâmica

Função malloc

- Use o tamanho correto

```
double *pd = (double*)malloc(10);
```

```
const int NUMERO_DE_DOUBLES = 10;
double *ptrd = (double*)malloc(NUMERO_DE_DOUBLES);
```

Alocação de Memória Dinâmica

Função malloc

- Use o tamanho correto

```
double *pd = (double*)m      (10);
```

```
const int NUMERO_     OUBLES
double *ptrd = (do                 NUMERO_DE_DOUBLES);
```

Alocação de Memória Dinâmica

Função calloc

- Esta função aloca e limpa a memória ao mesmo tempo;

```
void *calloc(size_t numElements, size_t elementSize);
```

Alocação de Memória Dinâmica

Função realloc `void *realloc(void *ptr, size_t size);`

- Utilizada quando é necessário aumentar ou diminuir a memória que foi alocada num ponteiro;
- Esta função retorna um ponteiro para um bloco de memória,
- Recebe dois argumentos, um ponteiro do bloco original e o novo tamanho;
  - O tamanho do bloco realocado será diferente do tamanho do bloco referenciado

Alocação de Memória Dinâmica

Função realloc

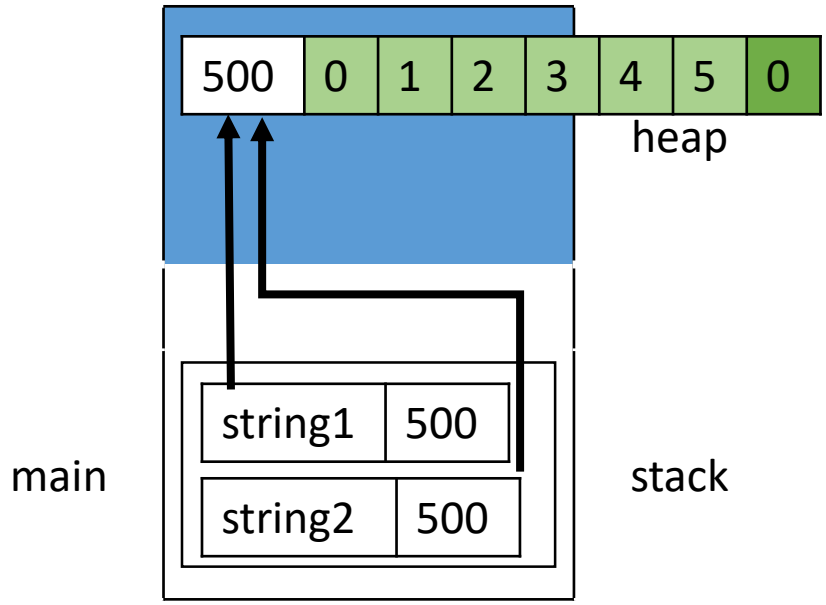
```
void *realloc(void *ptr, size_t size);
```

- O tamanho do bloco requisitado pode ser maior ou menor do que o bloco original
  - Se o tamanho é menor, o excesso de memória é retornado para o heap
  - Se o tamanho é maior que o bloco original, se for possível, a memória irá alocar numa região ligeiramente após o bloco original. Caso não seja possível, a memória será alocada numa nova região e o conteúdo do bloco antigo será copiado;
  - Se o tamanho é zero e o ponteiro é não nulo, o ponteiro será liberado (free)
  - Se o espaço não pode ser alocado, o bloco original não é mudado, e o ponteiro retornado é NULL

Alocação de Memória Dinâmica

Função realloc

- Diminuindo

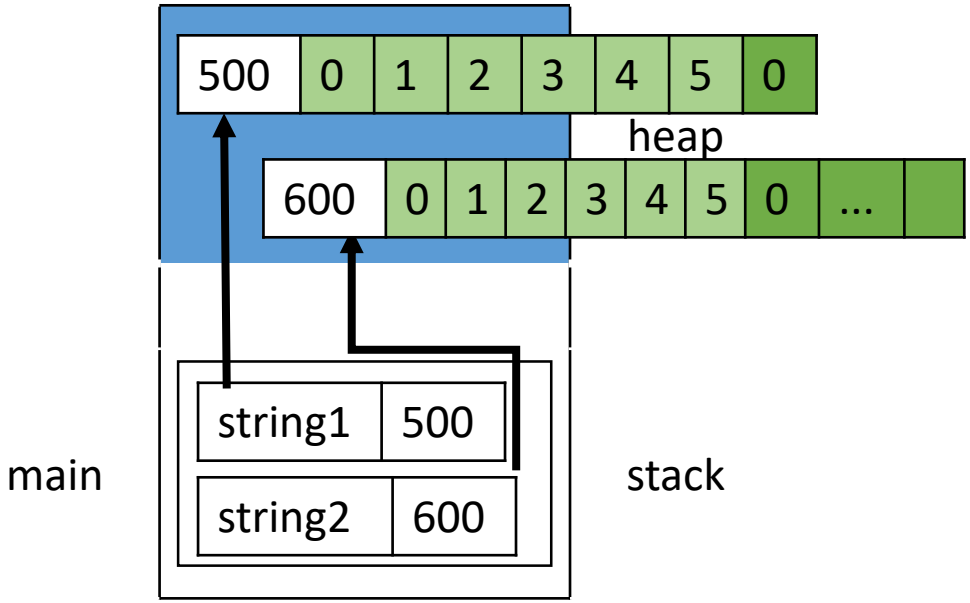



```
char *string1;
char *string2;
string1 = (char*) malloc(7);
strcpy(string1, "012345");

string2 = realloc(string1, 3);
printf("string1 Value: %p [%s]\n", string1, string1);
printf("string2 Value: %p [%s]\n", string2, string2);
```

Alocação de Memória Dinâmica

Função realloc

- Aumentando




```
char *string1;
char *string2;
string1 = (char*) malloc(7);
strcpy(string1, "012345");

string2 = realloc(string1, 12);
printf("string1 Value: %p [%s]\n", string1, string1);
printf("string2 Value: %p [%s]\n", string2, string2);
```

Alocação de Memória Dinâmica

Função free `void free(void *ptr);`

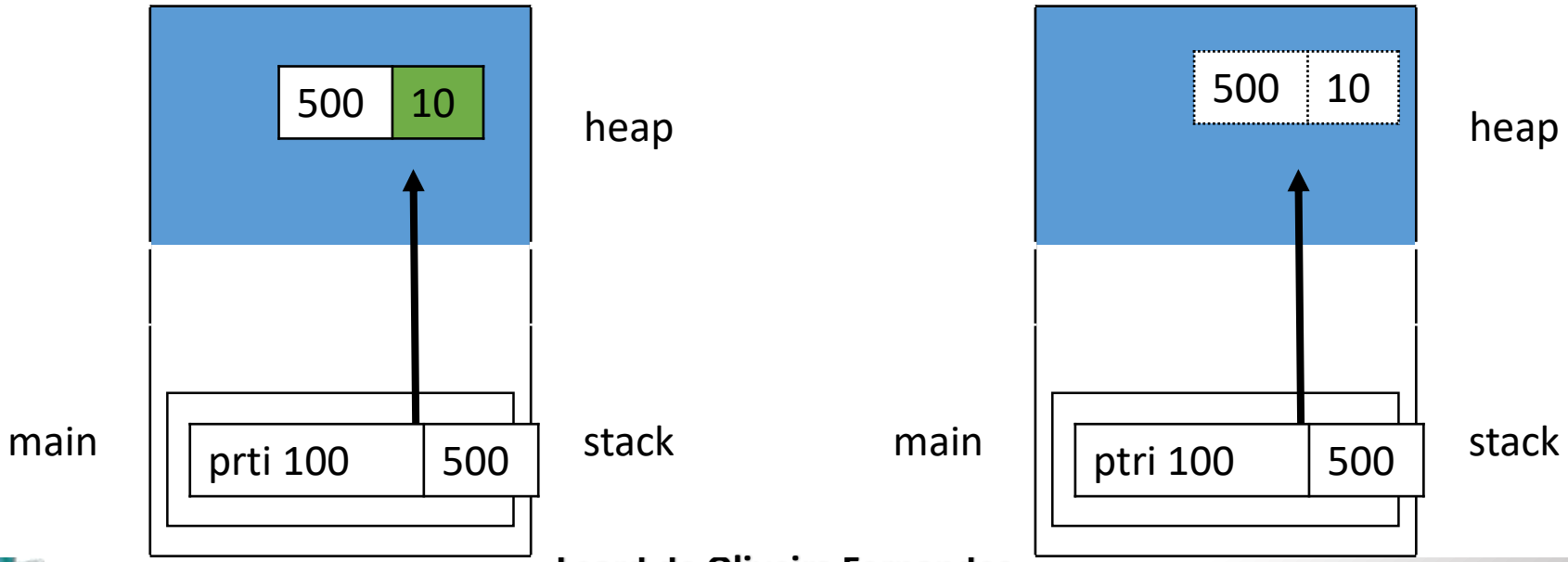
- Com a alocação de memória dinâmica, o programador é responsável por liberar a memória, quando ela não está mais sendo utilizada;
- Para isso, basta usar a função free;
- O Ponteiro deverá conter o endereço de memória alocado por uma função do tipo malloc;
- A memória irá retorna ao heap
  - O ponteiro poderá referenciar uma região, sempre assumindo que ele aponta para um lixo;
- A memória poderá ser realocado depois e povoada com diferentes dados

Alocação de Memória Dinâmica

Função free

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
free(ptri);
```

- O pontilhado na figura indica que a memória foi liberada...



Alocação de Memória Dinâmica

Função free

- O código abaixo garante que você não derefencie o ponteiro após a sua liberação
  - O comportamento não é definido

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
free(ptri);
ptri = NULL:
```

Alocação de Memória Dinâmica

Ponteiros Soltos

- Um ponteiro solto pode ocasionar os mais diversos tipos de problemas;
- Isto ocorre uma vez que utilizada a função free, o ponteiro continua referenciando a memória

```
int *ptri = (int*) malloc(sizeof(int));
*ptri = 10;
printf("%d\n", *ptri);
free(ptri);

*ptri = 30;
```

Alocação de Memória Dinâmica

Ponteiros Soltos

- O problema é pior quando temos dois ponteiros
  - Mais comum, durante cópias de ponteiros;

```
int *ptri1 = (int*) malloc(sizeof(int));
*ptri = 10;
int *ptri2;
printf("%d\n", *ptri);
ptri2 = ptri1;
free(ptri1);

*ptri2 = 30; //Ponteiro solto
```

Ponteiros e Funções

Leard de Oliveira Fernandes
CET 082

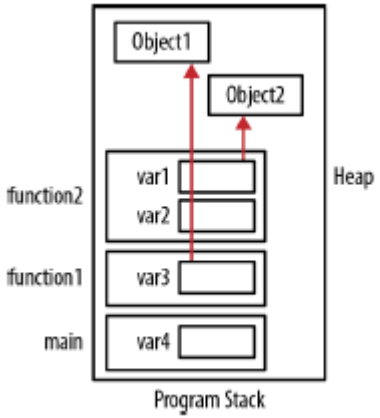
Ponteiros e Funções

- Já temos noção do poder dos ponteiros sobre uma função
  - Torna possível enviar dados que possam ser modificados por uma função;
  - Dados complexo também pode ser manipulados e/ou retornados de uma função na forma de um ponteiro para uma estrutura;
- Quando um ponteiro armazena o endereço de uma função, ele apresenta meios de manipular, dinamicamente, o fluxo de sua execução;
- Quando falamos sobre funções os ponteiros podem ser manipulados de duas formas
  - Passando um ponteiro para função
  - Declarando um ponteiro para uma função

Pilha e Heap do Programa

- A essa altura você já sabe que estamos falando de memória!!
- São elementos importantes no runtime system do C
  - Ambiente disponibilizado pelo SO;
  - Ele é responsável por estruturar o seu programa;
  - Em linha gerais, define o stack e o heap;

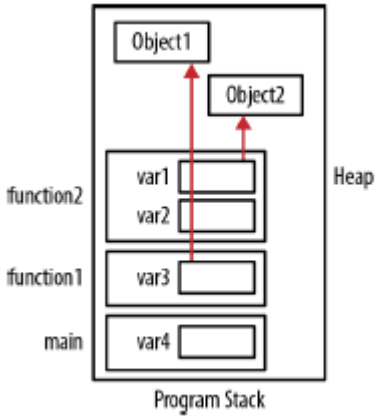
Pilha e Heap do Programa



- A pilha do programa é a área de memória que suporta as funções de execução
  - Normalmente é compartilhada com o heap
- A pilha do programa, normalmente, ocupa a parte mais inferior desta região de memória;
- Enquanto o heap, a parte superior...

```
void function2() {
    Object *var1 = ...;
    int var2;
    printf("Exemplo de Stack\n");
}
void function1() {
    Object *var3 = ...;
    function2();
}
int main() {
    int var4;
    function1();
}
```

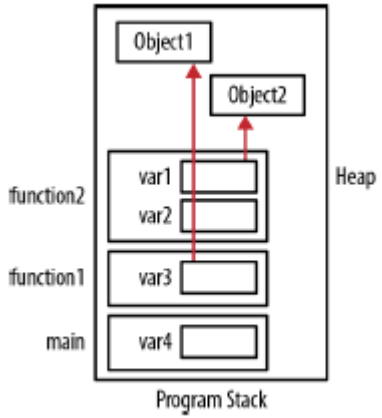
Pilha e Heap do Programa



- A pilha do programa armazena os quadros da pilha
  - Também conhecido por registros de ativação ou quadros de ativação
- Stack Frames armazenam os parâmetros e as variáveis locais de uma função;
- O heap gerencia ...?

```
void function2() {
    Object *var1 = ...;
    int var2;
    printf("Exemplo de Stack\n");
}
void function1() {
    Object *var3 = ...;
    function2();
}
int main() {
    int var4;
    function1();
}
```

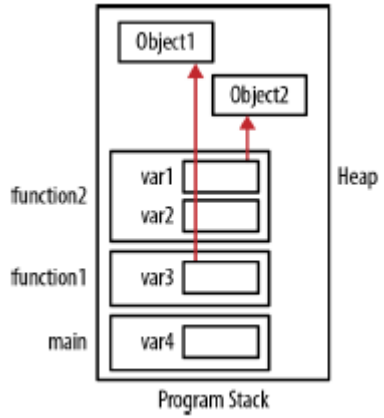
Pilha e Heap do Programa



- Conforme as funções são chamadas, seus stack frames são inseridos no topo da pilha do...
  - E a pilha cresce :-P
- Quando uma função termina, o seu stack frame é removido do topo da pilha do programa;
- A memória utilizada pelo stack frame não é limpa...
  - Eventualmente pode ser substituída por outro stack frame...

```
void function2() {
    Object *var1 = ...;
    int var2;
    printf("Exemplo de Stack\n");
}
void function1() {
    Object *var3 = ...;
    function2();
}
int main() {
    int var4;
    function1();
}
```

Pilha e Heap do Programa

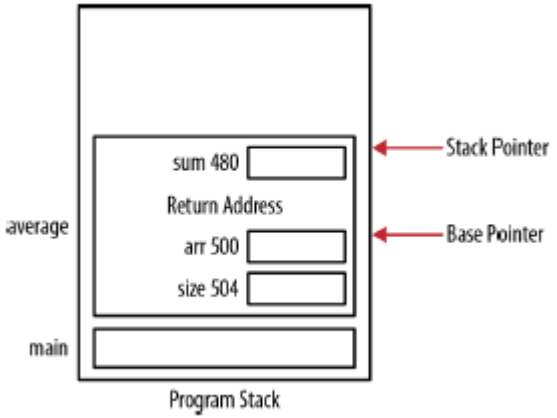


- Quando a memória é alocada dinamicamente ela vem do heap;
  - Cresce de cima para baixo
- O heap ficará fragmentado quando
  - Alocar memória, e
  - Desalocar memória
- Memória pode ser alocada em qualquer lugar no heap;

```
void function2() {
    Object *var1 = ...;
    int var2;
    printf("Exemplo de Stack\n");
}
void function1() {
    Object *var3 = ...;
    function2();
}
int main() {
    int var4;
    function1();
}
```

Organização de um stack frame

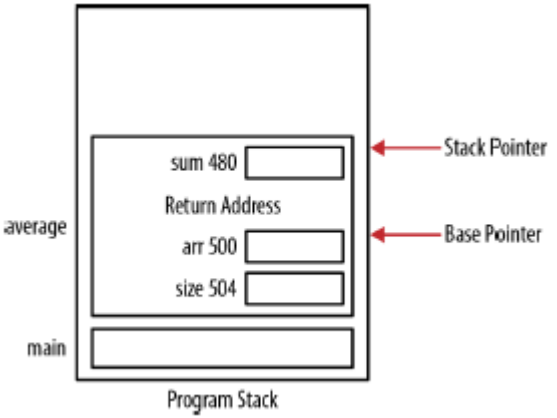
- O stack frame consiste de diversos elementos, incluindo:
  - Endereço de retorno: Endereço no programa, onde a função deve retornar ao terminar;
  - Armazenamento de dados locais: Memória alocada por variáveis locais;
  - Armazenamento de parâmetros: Memória utilizada para os parâmetros da função;
  - Ponteiros de pilha e base: Ponteiros utilizados pelo runtime system para gerencia a pilha




```
float average(int *arr, int size)
{
    int sum;
    printf("arr: %p\n",&arr);
    printf("size: %p\n",&size);
    printf("sum: %p\n",&sum);
    for(int i=0; i<size; i++) {
        sum += arr[i];
    }
    return (sum * 1.0f) / size;
}
```

Organização de um stack frame

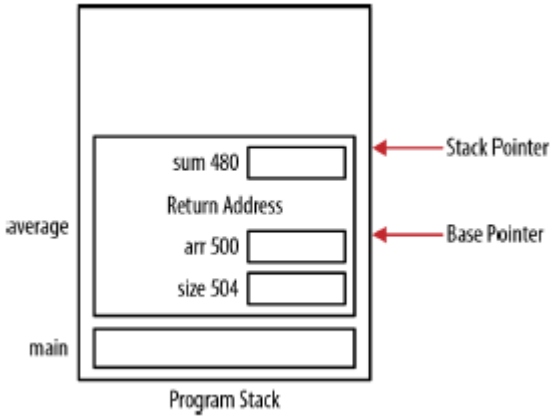
- O ponteiro da pilha aponta para o topo do stack frame
- O ponteiro de base, quando presente, aponta para um endereço dentro do stack frame;
  - Por exemplo o endereço de retorno;
  - É utilizado para acessar os elementos no quadro...
- Nenhum desses ponteiros são ponteiros do C;
  - São apenas endereços!!!




```
float average(int *arr, int size)
{
    int sum;
    printf("arr: %p\n",&arr);
    printf("size: %p\n",&size);
    printf("sum: %p\n",&sum);
    for(int i=0; i<size; i++) {
        sum += arr[i];
    }
    return (sum * 1.0f) / size;
}
```

Organização de um stack frame

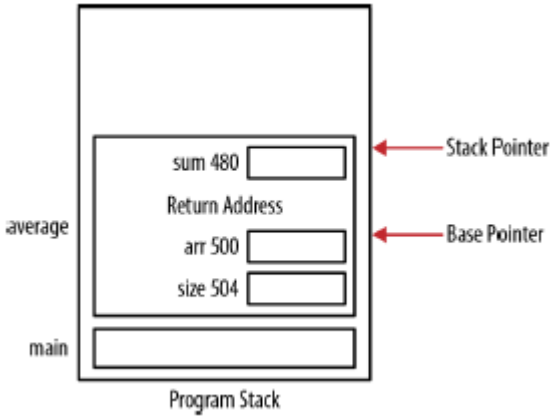
- No exemplo ao lado poderíamos ter os seguinte valores
  - arr: 0x500
  - size: 0x504
  - sum: 0x480
- O “buraco” entre os parâmetros e as variáveis locais é em função dos elementos do quadro da pilha utilizados para o seu gerenciamento;




```
float average(int *arr, int size)
{
    int sum;
    printf("arr: %p\n",&arr);
    printf("size: %p\n",&size);
    printf("sum: %p\n",&sum);
    for(int i=0; i<size; i++) {
        sum += arr[i];
    }
    return (sum * 1.0f) / size;
}
```

Organização de um stack frame

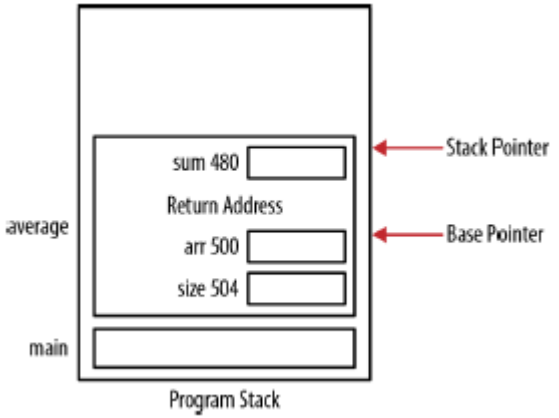
- E a variável i?
- Ela não é incluída no stack frame
- O C trata blocos de instruções como mini-funções;
  - Ou seja, um novo stack frame acima de average




```
float average(int *arr, int size)
{
    int sum;
    printf("arr: %p\n",&arr);
    printf("size: %p\n",&size);
    printf("sum: %p\n",&sum);
    for(int i=0; i<size; i++) {
        sum += arr[i];
    }
    return (sum * 1.0f) / size;
}
```

Organização de um stack frame

- Como os stack frames são inseridos na pilha do programa, o sistema pode ficar sem memória;
- Esta condição é chamada de estouro de pilha (stack overflow)
- Geralmente, termina o programa abruptamente;




```
float average(int *arr, int size)
{
    int sum;
    printf("arr: %p\n",&arr);
    printf("size: %p\n",&size);
    printf("sum: %p\n",&sum);
    for(int i=0; i<size; i++) {
        sum += arr[i];
    }
    return (sum * 1.0f) / size;
}
```

Passagem e Retorno por ponteiros

- A passagem de ponteiros permite o objeto referenciado ser acessível em múltiplas funções sem a necessidade de torná-lo global;
- Em outras palavras, somente as funções que precisam ter acesso ao objeto irão ter.
  - Sem a necessidade de duplicar o objeto;
- Se o dado necessita ser modificado por uma função, ele precisa ser passado por um ponteiro;

Passagem e Retorno por ponteiros

- Também podemos passar um dado por ponteiro e proibir que o mesma seja modificado
  - Passando um ponteiro para uma constante
- Quando o dado é um ponteiro, podemos passar um ponteiro para um ponteiro

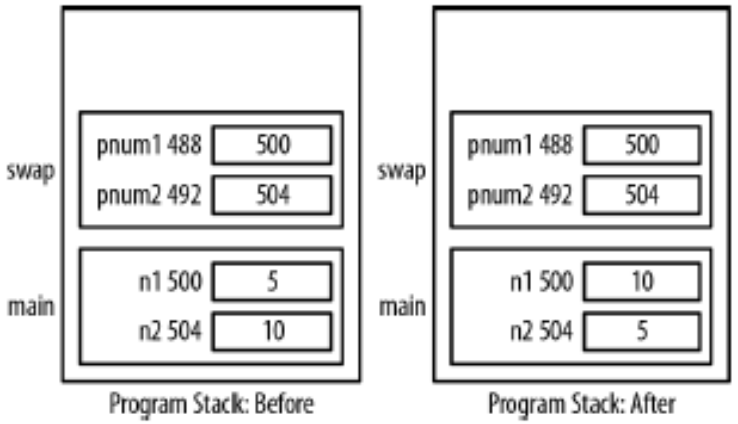
Passagem e Retorno por ponteiros

- Parâmetros, incluindo ponteiros, são passados por valor;
  - Ou seja, uma cópia do argumento é passado para a função
- O que é eficiente com uma estrutura de dados grande;

Passando dados utilizando ponteiros

- Uma das razões principais para passar dados utilizando um ponteiro é tornar a função capaz de modifica-los;

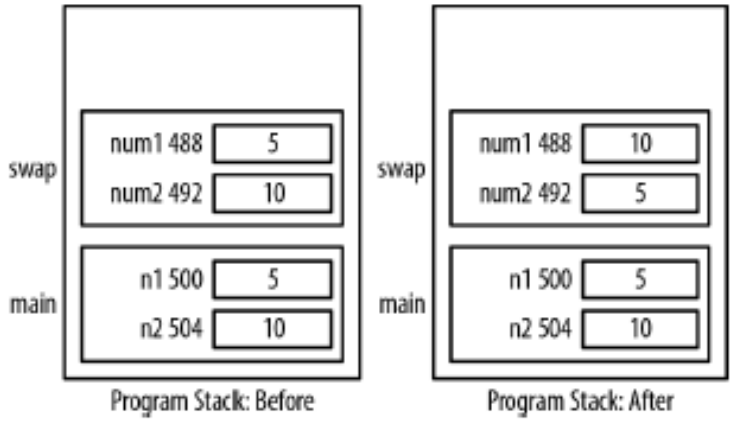
```
void swapP(int *pnum1, int *pnum2) {
    int temp = *pnum1;
    *pnum1=*pnum2;
    *pnum2=temp;
}

int main()
{
    int n1=5, n2=10;
    swapP(&n1, &n2);
    return 0;
}
```



Passando dados por valor

- Se não passamos dados por ponteiro, então não alteramos os dados

```
void swap(int pnum1, int pnum2) {
    int temp = pnum1;
    pnum1=pnum2;
    pnum2=temp;
}

int main()
{
    int n1=5, n2=10;
    swap(n1, n2);
    return 0;
}
```



Passando um Ponteiro para constante

- Técnica bastante comum quando queremos passar dados, sem cópia e com segurança;
- Quando não desejamos modificar os dados apontados, utilizamos um ponteiro para constante

```
void exemplo(const int *pnum1, int pnum2) {
    *pnum2=*pnum1;
}

int main()
{
    const int n1=5;
    int n2=10;
    exemplo(&n1, &n2);
    return 0;
}
```

Passando um Ponteiro para constante

- Técnica bastante comum quando queremos passar dados, sem cópia e com segurança;
- Quando não desejamos modificar os dados apontados, utilizamos um ponteiro para constante

```
void exemplo(const int *pnum1, int pnum2) {
    *pnum2=*pnum1;
}

int main()
{
    const int n1=5;
    int n2=10;
    exemplo(&n1, &n2);
    return 0;
}
```

Retornando um Ponteiro

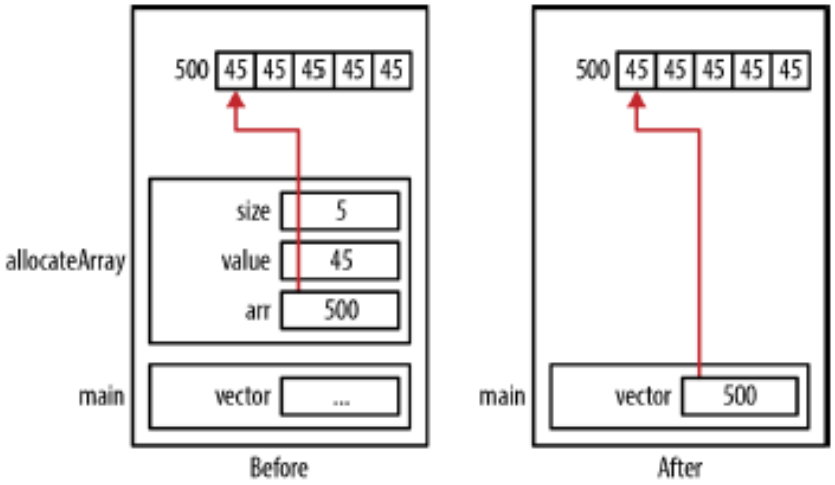
- Não é segredo!!
- Basta declarar o tipo de retorno para um ponteiro do tipo apropriado;
- Se precisamos retornar um objeto numa função, podemos utilizar duas técnicas:
  - Alocando a memória dentro da função utilizando a função malloc e retornando o seu endereço. O chamador é responsável por desalocar a memória retornada;
  - Passando o objeto para a função onde ele é modificado. O chamador fica responsável pela alocação e desalocação de memória do objeto

Retornando um Ponteiro

- Primeira técnica

```
int* allocArray(int size, int value) {
    int* arr = (int*)malloc(size * sizeof(int));
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

```
int* vector = allocArray(5,45);

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



Retornando um Ponteiro

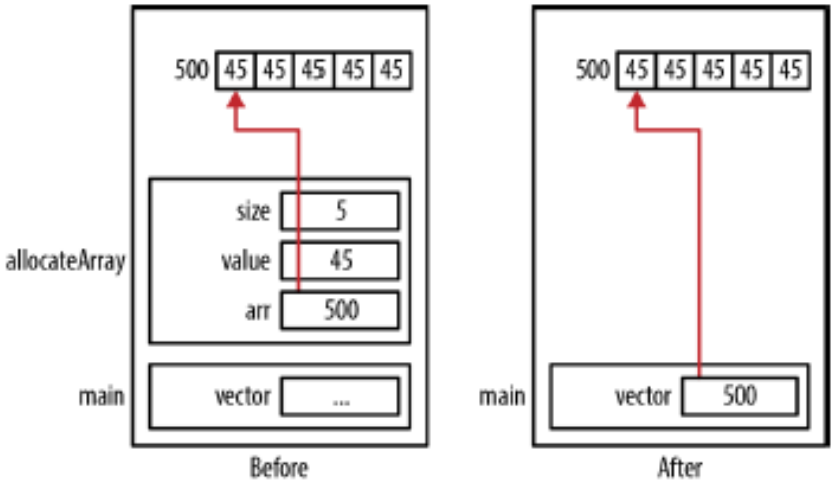
- Primeira técnica
- Não é recomendável retornar ponteiros!
- Apesar de funcionar corretamente, temos alguns problemas em potencial:
  - Retornar um ponteiro não inicializado;
  - Retornar um ponteiro para um endereço inválido;
  - Retornar um ponteiro para uma variável local;
  - Retornar um ponteiro e falhar em liberar a sua memória;
- No último caso, o chamador é responsável por liberar esta memória
  - Caso contrário, teremos um vazamento de memória

Retornando um Ponteiro

- Primeira técnica

```
int* allocArray(int size, int value) {
    int* arr = (int*)malloc(size * sizeof(int));
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

```
int* vector = allocArray(5,45);

for(int i=0; i<5; i++) {
    printf("%d\n", vector[i]);
}
free(vector);
```

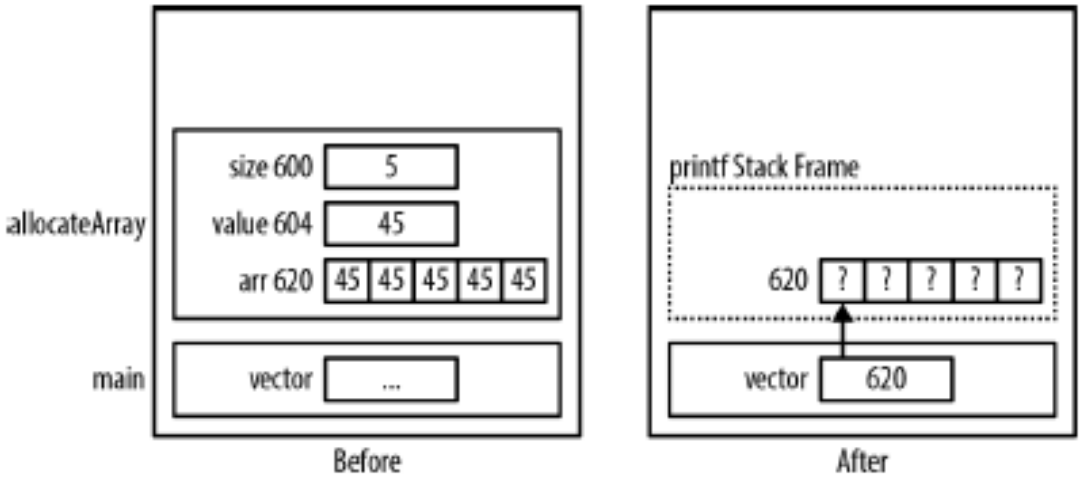


Retornando um Ponteiro

- Ponteiro para dados locais

```
int* allocArray(int size, int value) {
    int arr[size];
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

```
int* vector = allocArray(5,45);

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



Retornando um Ponteiro

- Possível Solução??

```
int* allocArray(int size, int value) {
    static int arr[5];
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

```
int* vector = allocArray(5,45);

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

- Possui utilidade quando queremos retornar mensagens de erro...

Retornando um Ponteiro

- Passando um ponteiro Nulo
- Nesta versão é passado um ponteiro para um array, com o seu tamanho e valor
- O ponteiro é retornado por conveniência...
- Esta versão não aloca memória

```
int* allocArray(int * arr, int size, int value) {
    if(arr != NULL) {
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
    return arr;
}
```

```
int* vector = (int*)malloc(5*sizeof(int));
allocArray(vector, 5, 45);
```

Ponteiro para Ponteiro

- Quando um ponteiro é passado para uma função, ele é passado por valor;
- Se queremos modificar o ponteiro original e não sua cópia, precisamos passar um ponteiro para ponteiro

Ponteiro para Ponteiro

- Neste exemplo é passado um ponteiro para um array de inteiro que:
  - Será associado a uma região de memória,
  - Inicializado
- A função irá retornar a memória alocada de volta, por meio do primeiro parâmetro;

```
void allocArray(int **arr, int size, int value) {
    *arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

```
int* vector = NULL;

allocArray(&vector, 5, 45);
```

Ponteiro para Ponteiro

- O endereço desta memória alocada é associada a um ponteiro para inteiro
- Para modificar este ponteiro na função, precisamos passar o seu endereço
- Assim, declaramos o parâmetro como ponteiro para ponteiro para um int;

```
void allocArray(int **arr, int size, int value) {
    *arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

```
int* vector = NULL;

allocArray(&vector, 5, 45);
```

Ponteiro para Ponteiro

- O endereço retornado por malloc é associado a arr
- Dereferenciando um ponteiro para um ponteiro para um inteiro, resulta num ponteiro para inteiro
  - O motivo disto é por conta do endereço do vetor, nós estamos modificando um vetor

```
void allocArray(int **arr, int size, int value) {
    *arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

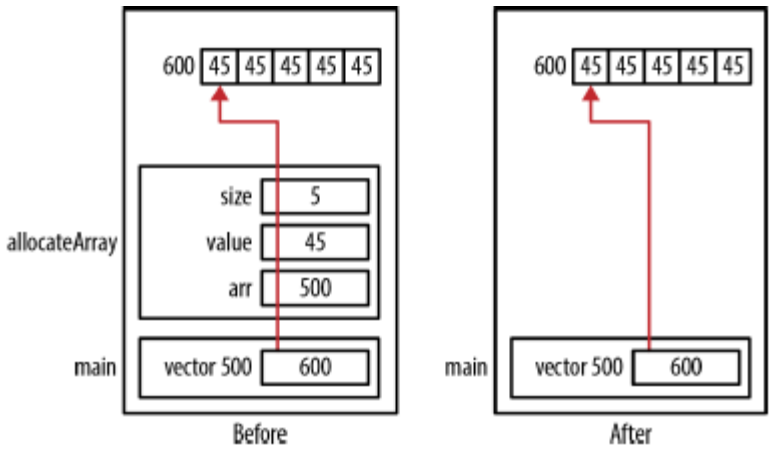
```
int* vector = NULL;

allocArray(&vector, 5, 45);
```

Ponteiro para Ponteiro

- O endereço retornado por malloc é associado a arr
- Dereferenciando um ponteiro para um ponteiro para um inteiro, resulta num ponteiro para inteiro
  - O motivo disto é por conta do endereço do vetor, nós estamos modificando um vetor

```
void allocArray(int **arr, int size, int value) {
    *arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

```
int* vector = NULL;

allocArray(&vector, 5, 45);
```



Ponteiro para Ponteiro

- E se implementarmos utilizando apenas um ponteiro, funciona?

```
void allocArray(int *arr, int size, int value) {
    arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

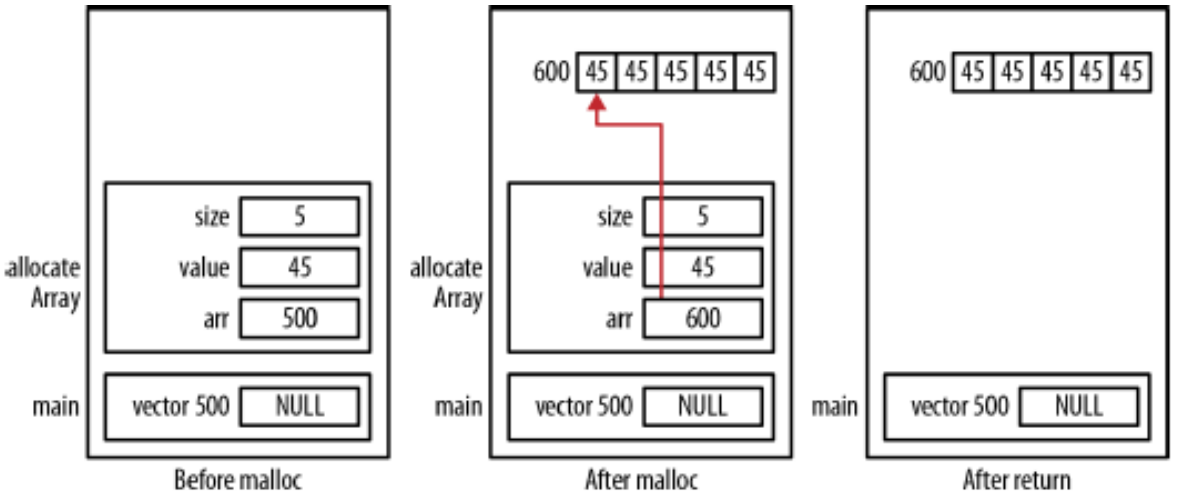
```
int* vector = NULL;

allocArray(&vector, 5, 45);
printf("%p\n",vector);
```

Ponteiro para Ponteiro

- E se implementarmos utilizando apenas um ponteiro, funciona? **Não**

```
void allocArray(int *arr, int size, int value) {
    arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL){
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

```
int* vector = NULL;

allocArray(&vector, 5, 45);
printf("%p\n",vector);
```



Escrevendo sua própria função free

- Diversos problemas podem ocorrer no simples uso da função free
  - Comportamentos inesperados
- A função free não checa se o ponteiro passado é nulo, e não seta o ponteiro com nulo...

Escrevendo sua própria função free

- A função abaixo apresenta um solução simples para executar o free
  - Requer o uso de um ponteiro para ponteiro;
- Seu parâmetro é declarado como um ponteiro para ponteiro para void
  - O que nos ajuda a modificar o ponteiro passado!
  - Usar void permite que todos os tipos de ponteiros sejam passados
  - Mas precisamos fazer cast, por conta dos warnings

```
void freeSeguro(void **pp){
    if(pp != NULL && *pp != NULL){
        free(*pp);
        *pp=NULL;
    }
}
```

Escrevendo sua própria função free

- Para resolver isso, basta criar uma macro com
  - O cast para void: (void**)
  - E o endereço do ponteiro: &(p)
- Desta forma, não precisamos a todo momento repetir estes comandos
- Basta usar **freeSeguro(ponteiro)**

```
void freeSeg(void **pp){
    if(pp != NULL && *pp != NULL){
        free(*pp);
        *pp=NULL;
    }
}
#define freeSeguro(p) freeSeg((void**)&(p))
```

Escrevendo sua própria função free

- Exemplo

```
void freeSeg(void **pp){
    if(pp != NULL && *pp != NULL){
        free(*pp);
        *pp=NULL;
    }
}
#define freeSeguro(p) freeSeg((void**)&(p))

int main() {
    int *pi;
    pi = (int*) malloc(sizeof(int));
    *pi = 5;
    printf("Antes: %p\n",pi);
    freeSeguro(pi);
    printf("Depois: %p\n",pi);
    freeSeguro(pi);
    return (EXIT_SUCCESS);
}
```

Ponteiros de Função

- O ponteiro de função é um ponteiro que contém o endereço de uma função;
- Esta possibilidade permite aos programadores uma outra forma de executar funções numa ordem que não é exatamente conhecida no tempo de compilação sem utilizar instruções condicionais;
- Em contrapartida, o seu programa terá a execução mais lenta
- Processador não poderá utilizar a predição dos saltos em conjunto com o pipeline

Ponteiros de Função

- No esquema de predição o processador inicia um salto acreditando que ele irá ser executado;
- Se ele “adivinha” o salto correto, então as instruções no pipeline não serão descartadas;
- Assim, quando utilizamos ponteiro de função podemos reduzir a performance de execução;

Declarando Ponteiros de Função

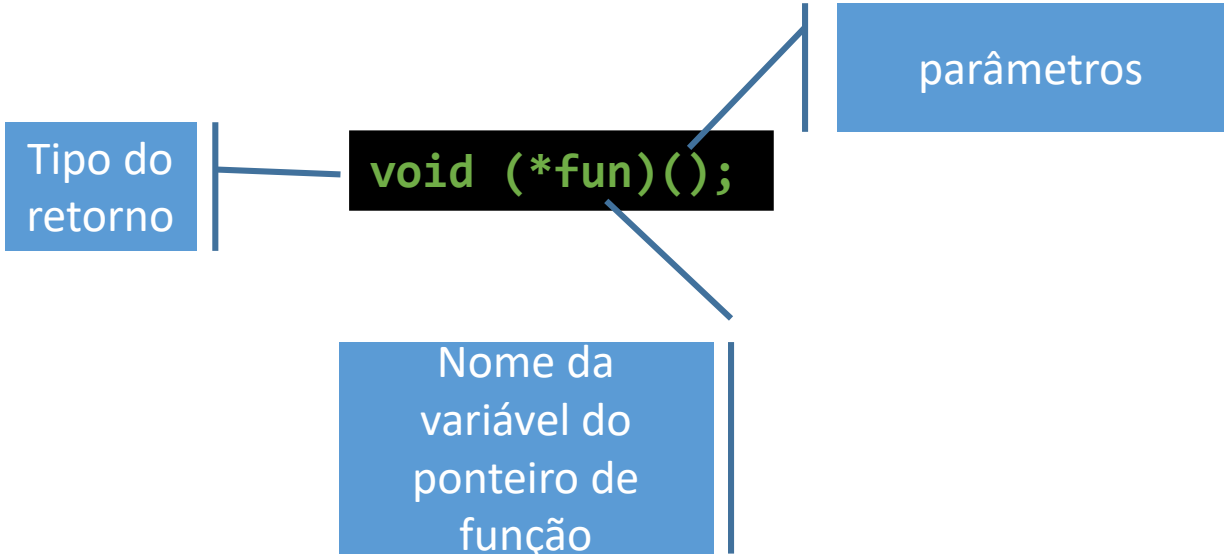
- Iniciamos com uma declaração simples

```
void (*fun)();
```

- Parece bastante com o protótipo de uma função
- Se removermos o primeiro conjunto de parênteses, temos o protótipo da função fun, que recebe um parâmetro void e retorna um ponteiro para void;
- Entretanto o parênteses a torna um ponteiro de função com o nome de fun;
- O asterisco indica que ela é um ponteiro

Declarando Ponteiros de Função

- Quando for utilizar, você deve assegurar que esteja correto, o C não checa se está certo ou não!!!



Declarando Ponteiros de Função

- Exemplos

```
int (*f1)(double);
void (*f2)(char*);
double* (*f3)(int, int);
```

Declarando Ponteiros de Função

- Exemplos

```
int (*f1)(double);        // Recebe um double e
                          // retorna um int

void (*f2)(char*);        // Recebe um ponteiro
                          //para char e retorna um void
double* (*f3)(int, int); //Recebe dois inteiros e
                          //retorna um ponteiro para double
```

- Dica: sempre comece o nome com ptrf ou fptr

Declarando Ponteiros de Função

- Não confundam um função que retorna ponteiro com uma que é ponteiro de função

```
int *f4();
int (*f5)();
int* (*f6)();
```

```
int* f4();
int (*f5)();
```

Utilizando Ponteiros de Função

- Abaixo temos um exemplo de ponteiro de função
- Definimos a função square, que retorna um inteiro elevado ao quadrado;
- Para utilizarmos o ponteiro de função para executar a função square, precisamos associá-lo ao endereço da função square;
- Basta atribuir o nome da função ao ponteiro de função;

```
int (*fptr1)(int);

int square(int num) {
    return num*num;
}
```

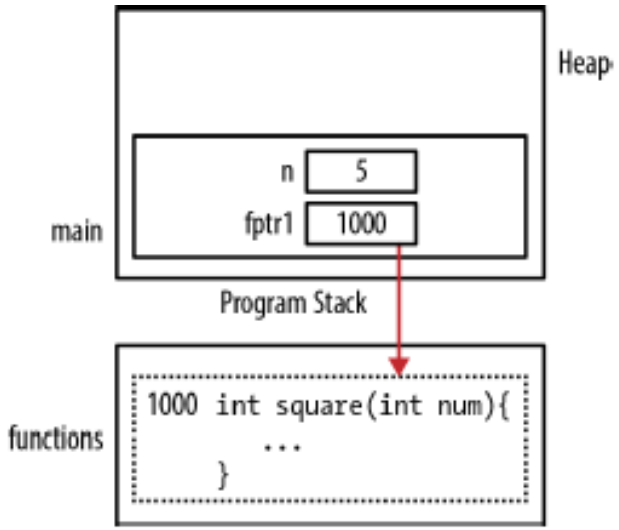
```
int n = 5;
fptr1 = square;
printf("%d squared is %d\n",n, fptr1(n));
```

Utilizando Ponteiros de Função

- Assim, podemos utilizar fptr1 como uma função;
- Quando executado irá apresentar o valor 25;

```
int (*fptr1)(int);

int square(int num) {
    return num*num;
}
```

```
int n = 5;
fptr1 = square;
printf("%d squared is %d\n",n, fptr1(n));
```



Utilizando Ponteiros de Função

- É comum declarar uma definição de tipo para ponteiros de função

```
typedef int (*funcptr)(int);

int square(int num) {
    return num*num;
}
```

```
int n = 5;
funcptr fptr2;
fptr2 = square;
printf("%d squared is %d\n",n, fptr2(n));
```

Passando Ponteiros de Função

- A passagem de ponteiros de função funciona como a passagem de ponteiros!
  - Basta usá-los como o parâmetro de uma função

```
int soma(int num1, int num2) {
    return num1 + num2;
}
int sub(int num1, int num2) {
    return num1 - num2;
}
typedef int (*fptrOperacao)(int, int);

int computar(fptrOperacao operacao, int num1, int num2){
    return operacao(num1,num2);
}
```

```
printf("%d\n", computar(soma, 10, 3));
printf("%d\n", computar(sub, 10, 3));
```

Retornando Ponteiros de Função

- Requer que você declare o tipo de retorno da função como ponteiro de função;
- Fica muito mais simples fazer isto usando a definição de tipo!

```
int soma(int num1, int num2) {
    return num1 + num2;
}
int sub(int num1, int num2) {
    return num1 - num2;
}
typedef int (*fptrOperacao)(int, int);

fptrOperacao seleciona(char codigo){
    switch(codigo){
        case '+': return soma;
        case '-': return sub;
    }
}

int avalia(char codigo, int num1, int num2){
    fptrOperacao operacao = seleciona(codigo);
    return operacao(num1, num2);
}
```

```
printf("%d\n", avalia('+', 10, 3));
printf("%d\n", avalia('-', 10, 3));
```

Array de Ponteiros de Função

- Podemos utilizar um array de ponteiros de função para selecionar uma função para ser executada com base em algum critério;
- Para isso, basta usar a declaração do ponteiro, como uma declaração de array;

```
typedef int (*ftrOperacao)(int, int);
ftrOperacao operacoes[128]={NULL};
```

```
int (*operacoes[128])(int, int)={NULL};
```

Array de Ponteiros de Função

- Neste exemplo podemos aceitar 128 diferentes operações

```
typedef int (* ftrOperacao)(int, int);
ftrOperacao operacoes[128]={NULL};
```

```
int (*operacoes[128])(int, int)={NULL};
```

```
void inicializaOperacoesArray() {
    operacoes['+'] = soma;
    operacoes['-'] = sub;
}
```

```
int avaliaArray(char codigo, int num1, int num2) {
    ftrOperacao operacao = operacoes[codigo];
    return operacao(num1, num2);
}
```

```
printf("%d\n", avaliaArray('+', 10, 3));
printf("%d\n", avaliaArray('-', 10, 3));
```

Comparando Ponteiros de Função

- Ponteiros de função podem ser comparados utilizando os operadores de igualdade e diferença!

```
ftrOperacao fptr1 = soma;
if(fptr1 == add) {
    printf("fptr1 aponta para a função soma\n");
} else {
    printf("fptr1 não aponta para a função soma\n");
}
```

Ponteiros e Estruturas

Leard de Oliveira Fernandes
CET 082

Ponteiros e Estruturas

Estruturas

- Estruturas no C podem representar os diferentes elementos de estrutura de dados
  - Nós de uma lista ligada, árvore, ...
- Os ponteiros formam a cola que mantém estes elementos ligados
- O entendimento de ponteiros é essencial para a criação de estruturas de dados complexas (e simples...)

Ponteiros e Estruturas

- As estruturas facilitam a organização dos dados
- Criar um array de entidades do tipo pessoa
  - Pessoa possui nome, sobrenome, rg, cpf, cnh, ctps, ...
- Caso você não utilize estruturas, será necessário criar um array para cada campo
- Entretanto, utilizando uma estrutura basta criar um array da estrutura composta por estes campos

Ponteiros e Estruturas

Estrutura

- Uma estrutura na linguagem C pode ser declarada de várias formas, iremos nos atentar a apenas duas
- Nosso foco será no seu uso com ponteiros

Ponteiros e Estruturas

Estrutura

- Na primeira forma declaramos a estrutura utilizando a palavra chave struct
- Utilizamos também o nome da estrutura préfixado, não é necessário, mas é utilizado como convenção de nomeação

```
struct _pessoa{
    char *nome;
    char *sobrenome;
    int rg;
    char *cnh;
    char *cpf;
    char *ctps;
};
```

Ponteiros e Estruturas

Estrutura

- Frequentemente utilizamos na declaração de uma struct a palavra chave typedef
  - Simplifica a sua utilização no programa
  - Define um novo tipo

```
typedef struct _pessoa{
    char *nome;
    char *sobrenome;
    unsigned int idade;
    char *rg;
    char *cnh;
    char *cpf;
    char *ctps;
} Pessoa;
```

Ponteiros e Estruturas

Estrutura

- Assim, a instância de uma pessoa é declarada como se segue

```
Pessoa pessoa;
```

- Alternativamente podemos declarar um ponteiro para uma Pessoa e alocar memória para ela

```
Pessoa *ptrPessoa;
ptrPessoa = (Pessoa *) malloc(sizeof(Pessoa));
```

Ponteiros e Estruturas

Estrutura

- Se utilizamos uma simples declaração de um estrutura, utilizamos a notação ponto (.) para acessar os campos nome e idade

```
Pessoa pessoa;
pessoa.nome = (char*) malloc(strlen("suzana“)+1);
strcpy(pessoa.nome, "suzana");
pessoa.idade = 29;
```

Ponteiros e Estruturas

Estrutura

- Entretanto, se utilizamos um ponteiro para uma estrutura, precisamos utilizar o operador aponta-para (->)

```
Pessoa *ptrPessoa;
ptrPessoa = (Pessoa*) malloc(sizeof(Pessoa))
ptrPessoa->nome = (char*) malloc(strlen("suzana“)+1);
strcpy(ptrPessoa->nome, "suzana");
ptrPessoa->idade = 29;
```

- Não precisamos utilizar o operador aponta-para, desde que dereferencie primeiro, e em seguida aplique o operador ponto

Ponteiros e Estruturas

Estrutura

- Mais trabalhosa, mas pode ser utilizada

```
Pessoa *ptrPessoa;
ptrPessoa = (Pessoa*) malloc(sizeof(Pessoa))
(*ptrPessoa).nome = (char*) malloc(strlen("suzana“)+1);
strcpy((*ptrPessoa).nome, "suzana");
(*ptrPessoa).idade = 29;
```

Ponteiros e Estruturas

Alocação da Memória

- O total de memória alocada para uma estrutura é pelo menos a soma dos tamanhos individuais de cada campo
- Entretanto, pode ser alocado um tamanho maior
- Isso ocorre quando é necessário alinhar certos tipos de dados regiões específicas;
  - Por exemplo, um short é alinhado em endereços divisíveis por 2, enquanto um int é alinhado em endereços divisíveis por 4

Ponteiros e Estruturas

Alocação da Memória

- Em função disso, é preciso ter cuidado ao
  - Realizar aritmética de ponteiros
  - Arrays de estruturas podem ter memória extra entre seus elementos

Ponteiros e Estruturas

```
typedef struct _pessoa{
    char *nome;
    char *sobrenome;
    unsigned int idade;
    char *rg;
    char *cnh;
    char *cpf;
    char *ctps;
} Pessoa;
```

Alocação da Memória

- No exemplo anterior, nossa estrutura teria 28 bytes
  - 4 bytes de cada ponteiro * 6
  - 4 bytes de um inteiro
- Caso o unsigned int seja substituído por short, teríamos 26 bytes
  - 4 bytes de cada ponteiro * 6
  - 2 bytes de um short
- Mas isso não ocorre
  - Nota: Sistemas 64 bits, temos 8 bytes para um ponteiro, o resultando em 56 bytes

```
typedef struct _outraPessoa{
    char *nome;
    char *sobrenome;
    short idade;
    char *rg;
    char *cnh;
    char *cpf;
    char *ctps;
} OutraPessoa;
```

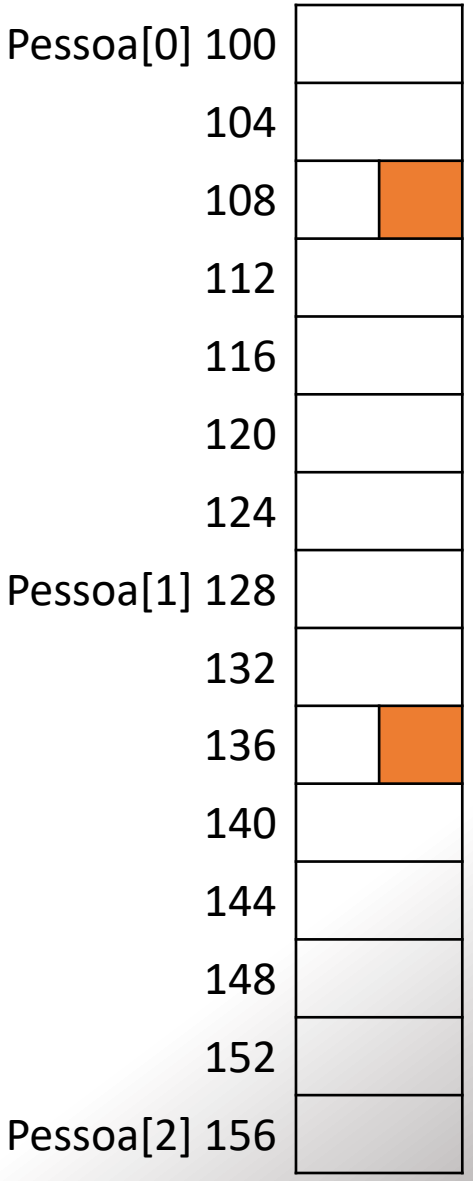
Ponteiros e Estruturas

Alocação da Memória

- Exemplo de um vetor para a estrutura OutraPessoa

```
typedef struct _outraPessoa{
    char *nome;
    char *sobrenome;
    short idade;
    char *rg;
    char *cnh;
    char *cpf;
    char *ctps;
} OutraPessoa;
```

```
OutraPessoa Pessoa[3];
```



Ponteiros e Estruturas

Desalocando estruturas

- Quando a memória é alocada para uma estrutura, o sistema de tempo de execução não irá alocar automaticamente memória para qualquer ponteiro definido dentro dela;
- Assim, quando uma estrutura de dados terminar, o runtime system não irá desalocar automaticamente a memória associada a cada ponteiro;

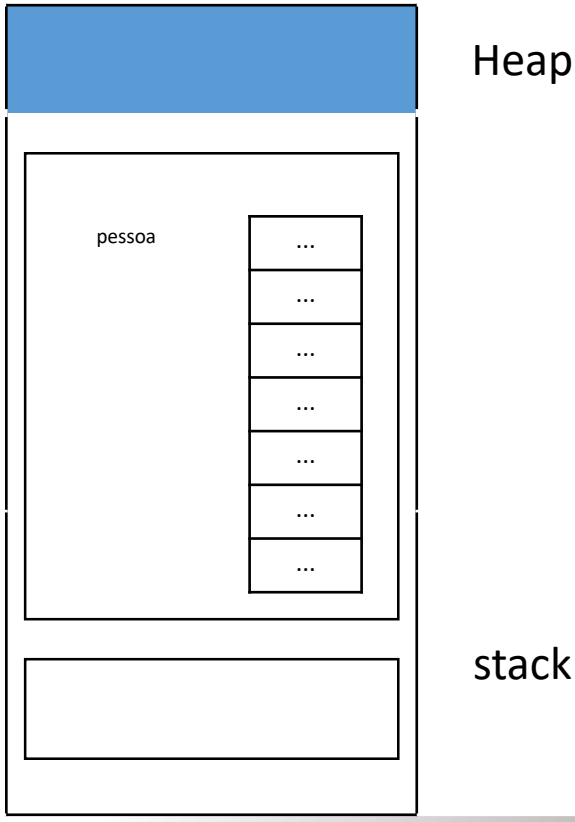
Ponteiros e Estruturas

```
typedef struct _pessoa{
    char *nome;
    char *sobrenome;
    unsigned int idade;
    char *rg;
    char *cnh;
    char *cpf;
    char *ctps;
} Pessoa;
```

Desalocando estruturas

- Considerando a estrutura
- Quando declaramos uma variável deste tipo ou alocamos memória dinâmica para este tipo, os ponteiros irão conter lixo

```
void processaPessoa(){
    Pessoa pessoa;
    …
}
```



Ponteiros e Estruturas

Desalocando estruturas

- Assim, durante a inicialização da estrutura cada campo deve possuir um valor associado

```
void inicializaPessoa(Pessoa *pessoa, const *char n, const char *s, uint i,
                      const char *rg, const char *cnh, const char *cpf,
                      consy char *ctps){
    pessoa->nome = (char*)malloc(strlen(n)+1);
    strcpy(pessoa->nome, n);
    pessoa->sobrenome = (char*)malloc(strlen(s)+1);
    strcpy(pessoa->sobrenome, s);
    pessoa->idade = i;
    pessoa->rg = (char*)malloc(strlen(rg)+1);
    strcpy(pessoa->rg, rg);
    pessoa->cnh = (char*)malloc(strlen(cnh)+1);
    strcpy(pessoa->cnh, cnh);
    pessoa->cpf = (char*)malloc(strlen(cpf)+1);
    strcpy(pessoa->cpf, cpf);
    pessoa->ctps = (char*)malloc(strlen(ctps)+1);
    strcpy(pessoa->ctps, ctps);
}
```

Ponteiros e Estruturas

Desalocando estruturas

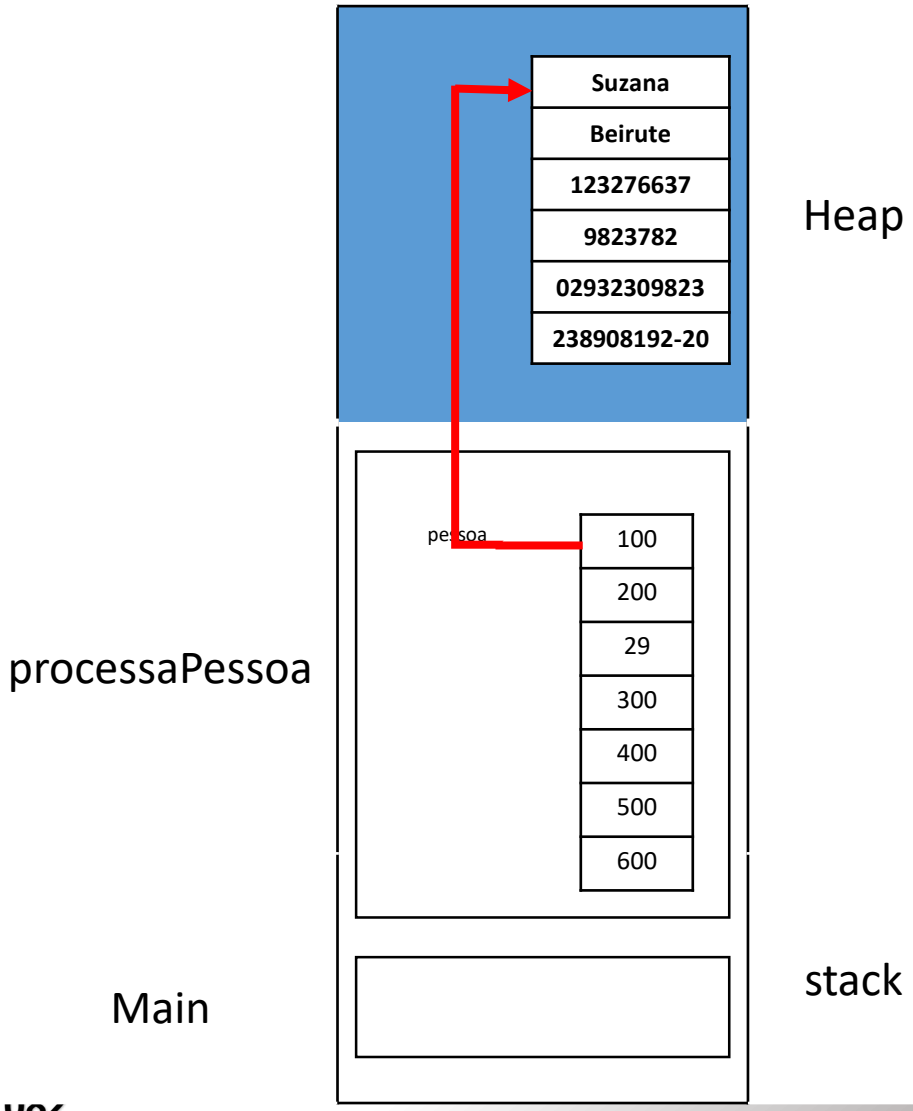
- Podemos então utilizar a função anterior

```
void processaPessoa(){
    Pessoa pessoa;
    inicializaPessoa(&pessoa, "Suzana", "Beirute", 29, "123276637“, “9823782”,
                     “02932309823”, “238908192-20”);
    ...
}
int main(){
    processaPessoa();
}
```

Ponteiros e Estruturas

Desalocando estruturas

- Estado da memória



Ponteiros e Estruturas

Desalocando estruturas

- Assim quando a função retornar, a memória para pessoa irá “embora” (não será mais utilizada)
- Entretanto as strings alocadas dinamicamente não serão liberadas;
- Como perdemos o seu endereço, não podemos liberar....
- Assim, precisamos liberar a memória antes da função terminar

Ponteiros e Estruturas

```
void desalocaPessoa(Pessoa *pessoa){
    free(pessoa->nome);
    free(pessoa->sobrenome);
    free(pessoa->rg);
    free(pessoa->cnh);
    free(pessoa->cpf);
    free(pessoa->ctps);
}
```

Desalocando estruturas

- Esta função irá liberar a memória que foi alocada dinamicamente;
- Deve ser chamada antes do término da função

```
void processaPessoa(){
    Pessoa pessoa;
    inicializaPessoa(&pessoa, "Suzana", "Beirute", 29, "123276637“, “9823782”,
                     “02932309823”, “238908192-20”);
    ...
    desalocaPessoa(&pessoa);
}
int main(){
    processaPessoa();
}
```

Ponteiros e Estruturas

```
void desalocaPessoa(Pessoa *pessoa){
    free(pessoa->nome);
    free(pessoa->sobrenome);
    free(pessoa->rg);
    free(pessoa->cnh);
    free(pessoa->cpf);
    free(pessoa->ctps);
}
```

Desalocando estruturas

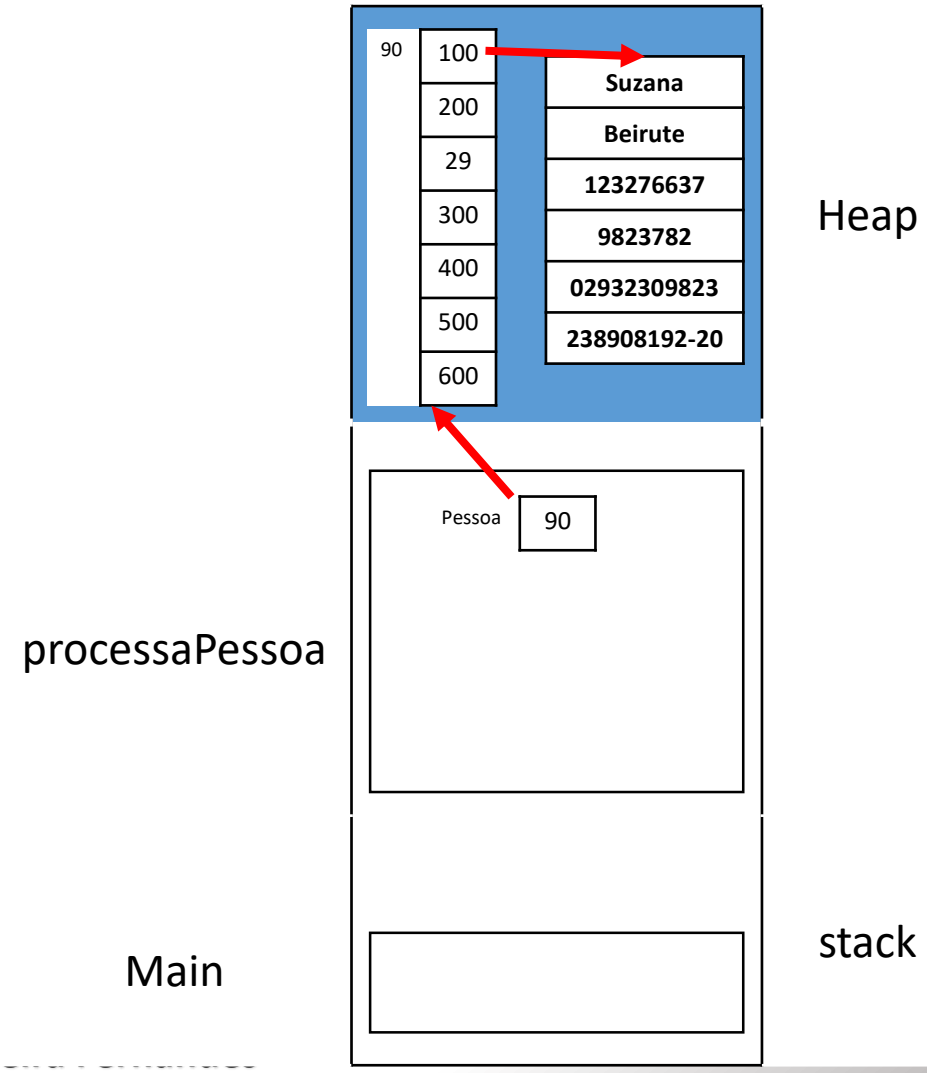
- Caso seja utilizado um ponteiro para pessoa, devemos liberar também esta memória

```
void processaPessoa(){
    Pessoa *pessoa;
    pessoa = (Pessoa *)malloc(sizeof(Pessoa));
    inicializaPessoa(pessoa, "Suzana", "Beirute", 29, "123276637“, “9823782”,
                     “02932309823”, “238908192-20”);
    ...
    desalocaPessoa(pessoa);
    free(pessoa);
}
int main(){
    processaPessoa();
}
```

Ponteiros e Estruturas

Desalocando estruturas

- Estado da memória



Ponteiros e Strings

Leard de Oliveira Fernandes
CET 082

Ponteiros e Strings

- Strings podem ser alocadas em diferentes regiões de memória;
- Os ponteiros são utilizados para realizar operações sobres as strings;
- No C, strings são regularmente passadas e retornadas para funções como ponteiros para char;
- Podemos passar como um ponteiro para um char ou como um ponteiro para uma constante char;

Ponteiros e Strings

- Uma string é uma sequência de caracteres terminado com o caractere ASCII NUL;
  - \0
- Nem todo array de caracteres é uma string!!
- São comumente armazenadas em array ou na memória alocada do Heap;

Ponteiros e Strings

- Existem dois tipos de strings no C:
  - Byte String
    - Sequencia de caracteres do tipo char
  - Wide String
    - Sequencia de caracteres do tipo wchar_t
- O tipo wchar_t é utilizado para caracteres expandidos
  - Podendo ter 16 ou 32 bits de largura;
- Ambas são finalizadas com o caractere \0;
- As funções para
  - Byte String estão na biblioteca string.h
  - Wide String estão na biblioteca wchar.h

Ponteiros e Strings

- O tamanho de uma string, é relativo a sua quantidade de caracteres, não contendo o caractere \0;
- Lembre-se:
  - NULL e NUL são conceitos diferentes!!
- Caracteres constantes são sequencias de caracteres entre aspas simples;
  - Pode conter mais de um caractere, no caso, quando utilizado as sequencia de escape

```
printf("%d“, sizeof(char));
printf("%d“, sizeof('b'));
```

Declarando Strings

- Array de caracteres com 32 posições

```
char cabecalho[32];
```

- Uma string requer um caracteres NUL, assim, este array pode armazenar 31 elementos da string;

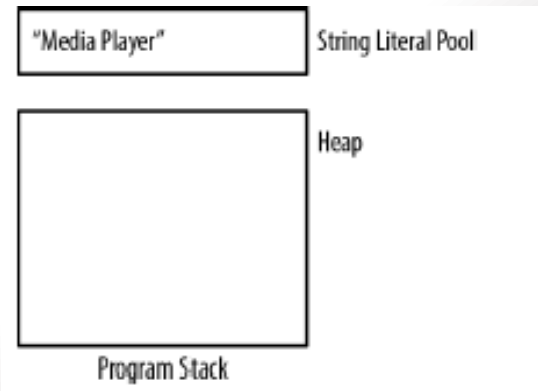
- Abaixo, temos um ponteiro para caractere;

```
char *cabecalho;
```

- Desde que ele não esteja inicializado, ele não referencia uma string;

O Pool de Literais

- Quando os literais são definidos, em geral são associados a um Pool de literais;
- O pool é uma área de memória que armazena uma sequencia de caracteres, que compõem uma string;
- Quando um literal é utilizado mais de uma vez, normalmente existe uma única cópia da string, no pool de literais;
- A ideia é reduzir o volume de memória utilizada;
- O GCC oferece uma opção para desligar o pool
  - -fwritable-strings



Quando um literal não é constante

- Em alguns compiladores, uma string literal é tradada como constante;
- Assim, não é possível modificá-la;
- No GCC, a modificação de uma string literal é possível;

```
char *str = “SOM”;
*str = ‘L’;
printf("%s\n", str);
```

- Caso queira garantir que não exista modificação

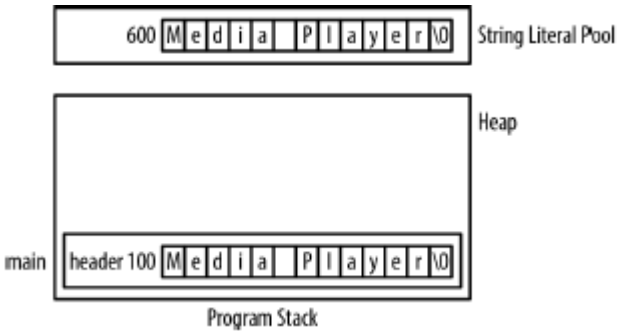
```
const char *str = “SOM”;
*str = ‘L’;
printf("%s\n", str);
```

Inicializando Strings

Podemos utilizar duas abordagens:

- Inicializando um array de caracteres
  - Usando o operador de inicialização

```
char cabecalho[] = “Media Player”;
```



  - Usando a função strcpy

```
char cabecalho[13];
strcpy(cabecalho, “Media Player”);
```

  - Ou....

```
char cabecalho[13];
cabecalho[0] =‘M’;
cabecalho[1] =‘e’;
…
cabecalho[12] =‘\0’;
```

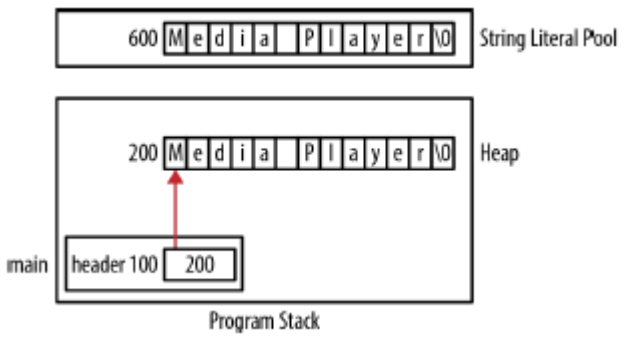
Inicializando Strings

- Podemos utilizar duas abordagens:
- Inicializando um ponteiro para um char

```
char *cabecalho;
```

```
char * cabecalho = (char*) malloc(strlen("Media Player")+1);
strcpy(cabecalho,"Media Player");
```

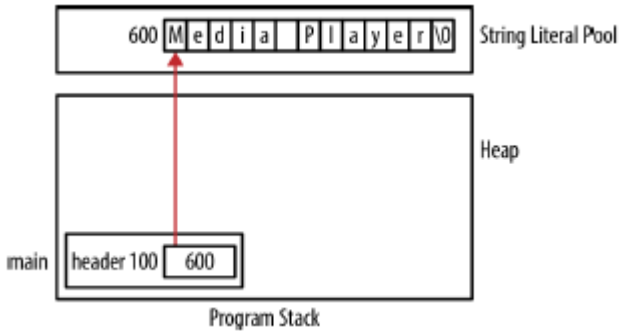
```
char * cabecalho = (char*) malloc(13);

*(cabecalho+1) ='M';
*(cabecalho+2) ='e';
…
*(cabecalho+12) ='\0';
```



Inicializando Strings

- Podemos utilizar duas abordagens:
- Inicializando um ponteiro para um char

```
char *cabecalho = "Media Player";
```



Leard de Oliveira Fernandes
CET 082

Inicializando Strings

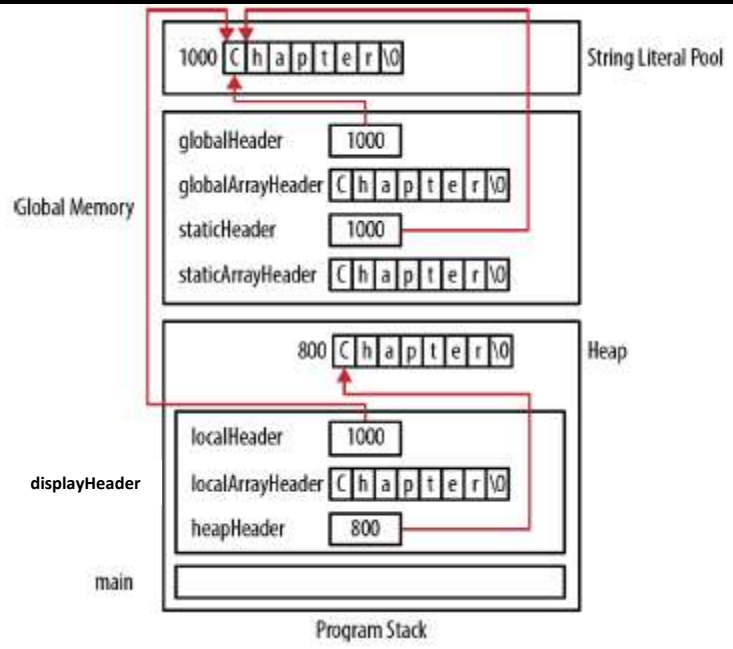
- Podemos utilizar duas abordagens:
- Inicializando um ponteiro para um char

```
char* prefix = '+'; // Errado
```

```
prefix = (char*)malloc(2);
*prefix = '+';
*(prefix+1) = 0; //*(prefix+1)=‘\0’
```

Strings e Memória

```
char* globalHeader = "Chapter";
char globalArrayHeader[] = "Chapter";
void displayHeader() {
   static char* staticHeader = "Chapter";
   char* localHeader = "Chapter";
   static char staticArrayHeader[] = "Chapter";
   char localArrayHeader[] = "Chapter";
   char* heapHeader = (char*)malloc(strlen("Chapter")+1);
   strcpy(heapHeader,"Chapter");
}
```



Operações Strings

- Comparação

```
int strcmp(const char *s1, const char *s2);
```

  - Negativo: s1 precede s2
  - Zero: iguais
  - Positivo: s2 precede s1

Operações Strings

- Copia

```
char* strcpy(char *s1, const char *s2);
```

  - Copia s2 para s1

Operações Strings

- Concatenar

```
char *strcat(char *s1, const char *s2);
```

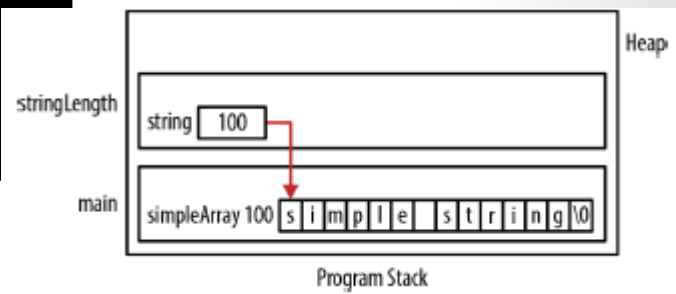
  - Une s2 ao conteúdo de s1

Passando Strings

- Não muda o conceito de passagem de parâmetros

```
size_t stringLength(char* string) {
    size_t length = 0;
    while(*(string++)) {
        length++;
    }
    return length;
}
```

```
char simpleArray[] = "simple string";
char *simplePtr = (char*)malloc(strlen("simple string")+1);
strcpy(simplePtr, "simple string");

printf("%d\n",stringLength(simplePtr));

printf("%d\n",stringLength(simpleArray)); //1
printf("%d\n",stringLength(&simpleArray)); //2
printf("%d\n",stringLength(&simpleArray[0])); //3
```



Passando Strings

- Para um const...

```
size_t stringLength(const char* string) {
    size_t length = 0;
    while(*(string++)) {
        length++;
    }
    return length;
}
```

```
size_t stringLength(const char* string) {
    …
    *string = 'A';
    …
}
```

Passando Strings

- Para ser inicializada
  - Passamos o buffer
  - O chamador é responsável pelo free
  - A função retorna um ponteiro para o buffer

```
char* format(char *buffer, size_t size, const char* name, size_t quantity, size_t weight) {
    snprintf(buffer, size, "Item: %s Quantity: %u Weight: %u", name, quantity, weight);
    return buffer;
}
```

Passando Strings

- Para ser inicializada
  - Passamos o buffer
  - O chamador é responsável pelo free
  - A função retorna um ponteiro para o buffer

```
char* format(char *buffer, size_t size, const char* name, size_t quantity, size_t weight) {
    char *formatString = "Item: %s Quantity: %u Weight: %u";
    size_t formatStringLength = strlen(formatString)-6;
    size_t nameLength = strlen(name);
    size_t length = formatStringLength + nameLength + 10 + 10 + 1;
    if(buffer == NULL) {
        buffer = (char*)malloc(length);
        size = length;
    }
    snprintf(buffer, size, "Item: %s Quantity: %u Weight: %u", name, quantity, weight);
    return buffer;
}
```

Passando Agumentos para uma App

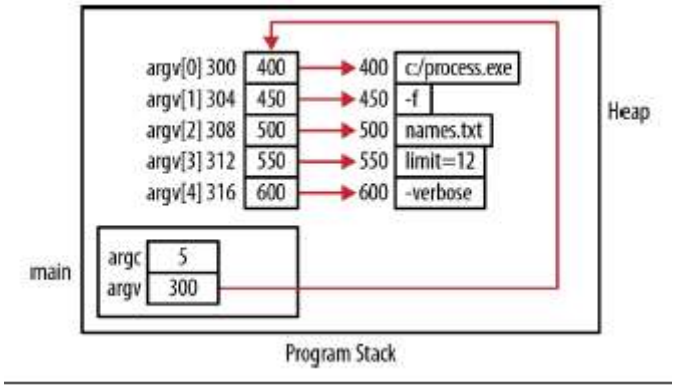
- A função main é normalmente a primeira função ser executada numa aplicação;
- É possível passar informações no console...
- O C utiliza os parâmetros argc e argv
  - Argc é um inteiro que indica quantos argumentos foram passados
  - Argv é um array de ponteiros de strings
- Pelo menos um argumento é passado...

```
int main(int argc, char** argv) {
    for(int i=0; i<argc; i++) {
        printf("argv[%d] %s\n",i,argv[i]);
    }
    ...
}
```

Passando Agumentos para uma App

- Exemplo

```
>process.exe -f names.txt limit=12 –verbose
argv[0] c:/process.exe
argv[1] -f
argv[2] names.txt
argv[3] limit=12
argv[4] -verbose
```



Retornando String

- Quando uma função retorna uma string ela retorna o seu endereço;
- O ponto principal é retornar uma string válida;
- Para isso, podemos retornar uma referência para:
  - Um literal (Pool)
  - Uma memória alocada dinamicamente
  - A uma variável local de string

Retornando o endereço de um literal

- Cuidado com os literais, pois nem sempre são tradados como constantes;
- Pode existir problemas quando existem nomes iguais;

```
char* retornaUmLiteral(int codigo) {
        switch(codigo) {
           case 100:
               return “UESC";
           case 200:
               return “UESB";
           case 300:
               return “UNEB";
           case 400:
               return “IFBA";
        }
}
```

Retornando o endereço de um literal

- Assim, é possível resolver possíveis problemas
  - Caso exista os mesmos literais no programa, e
  - Existam modificações

```
char* retornaUmLiteral(int codigo) {
        static char * uesc = "UESC";
        static char * uesb = "UESB";
        static char * uneb = "UNEB";
        static char * ifba = "IFBA";

        switch(codigo) {
           case 100:
               return uesc;
           case 200:
               return uesb;
           case 300:
               return uneb;
           case 400:
               return ifba;
        }
}
```

Retornando o endereço de um literal

- Retornar um ponteiro para uma string estática pode ser problemático

```
char* staticFormat(const char* nome, size_t quantidade, size_t peso) {
    static char buffer[64]; // Grande o suficiente
    sprintf(buffer, "Item: %s Quantidade: %u Peso: %u", nome, quantidade, peso);
    return buffer;
}
```

```
char* part1 = staticFormat(“Arroz",25,45);
char* part2 = staticFormat(“Feijao",55,5);
printf("%s\n",part1);
printf("%s\n",part2);
```

Retornando o endereço MD

- Se um string precisa ser retornada de uma função, a memória da string pode ser alocada da memória heap;

```
char* blanks(int number) {
    char* spaces = (char*) malloc(number + 1);
    int i;
    for (i = 0; i<number; i++) {
        spaces[i] = ' ';
    }
    spaces[number] = '\0';
    return spaces;
}
```

```
char *tmp = blanks(5);
```

```
printf("[%s]\n",blanks(5));
```

Retornando o endereço MD

- Se um string precisa ser retornada de uma função, a memória da string pode ser alocada da memória heap;

```
char* blanks(int number) {
    char* spaces = (char*) malloc(number + 1);
    int i;
    for (i = 0; i<number; i++) {
        spaces[i] = ' ';
    }
    spaces[number] = '\0';
    return spaces;
}
```

```
char *tmp = blanks(5);
```

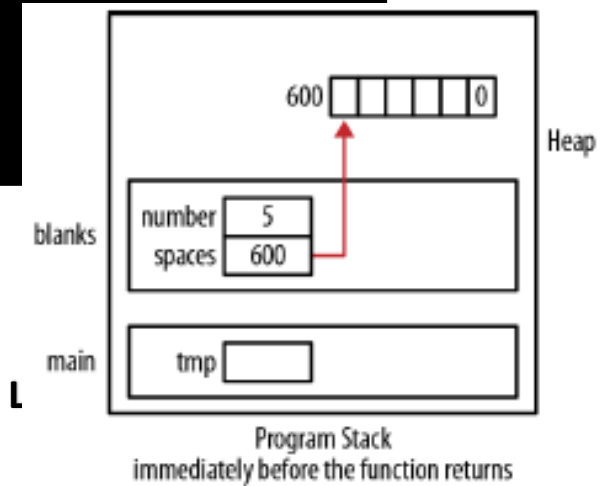
```
char *tmp = blanks(5);
printf("[%s]\n",tmp);
free(tmp);
```

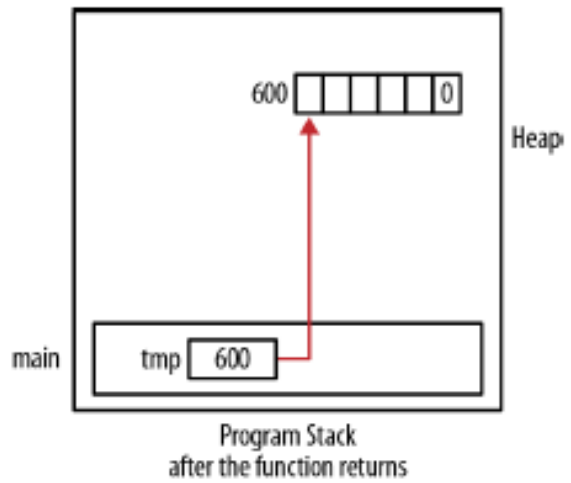
Retornando o endereço MD

- Se um string precisa ser retornada de uma função, a memória da string pode ser alocada da memória heap;

```
char* blanks(int number) {
    char* spaces = (char*) malloc(number + 1);
    int i;
    for (i = 0; i<number; i++) {
        spaces[i] = ' ';
    }
    spaces[number] = '\0';
    return spaces;
}
```

```
char *tmp = blanks(5);
```





Ponteiro de função e Strings

- Flexibilidade

```
typedef int (fptrOperation)(const char*, const char*);
```

```
int compare(const char* str1, const char* str2) {
    return strcmp(str1,str2);
}
int compareCBaixa(const char* str1, const char* str2) {
    char* t1 = stringCBaixa(str1); //Retorna do heap
    char* t2 = stringCBaixa(str2); //Retorna do heap
    int resultado = strcmp(t1, t2);
    free(t1);
    free(t2);
    return resultado;
}
char* stringBaixa(const char* string) {
    char *tmp = (char*) malloc(strlen(string) + 1);
    char *start = tmp;
    while (*string != 0) {
        *tmp++ = tolower(*string++);
    }
    *tmp = 0;
    return start; //Retorna do heap
}
```

Ponteiro de função e Strings

- Flexibilidade

```
typedef int (fptrOperacao)(const char*, const char*);
```

```
void sort(char *array[], int size, fptrOperacao operacao) {
    int swap = 1;
    while(swap) {
        swap = 0;
        for(int i=0; i<size-1; i++) {
            if(operacao(array[i],array[i+1]) > 0){
                swap = 1;
                char *tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
            }
        }
    }
}
```

```
void mostraNomes(char* nomes[], int size) {
    for(int i=0; i<size; i++) {
        printf("%s ",nomes[i]);
    }
    printf("\n");
}
```

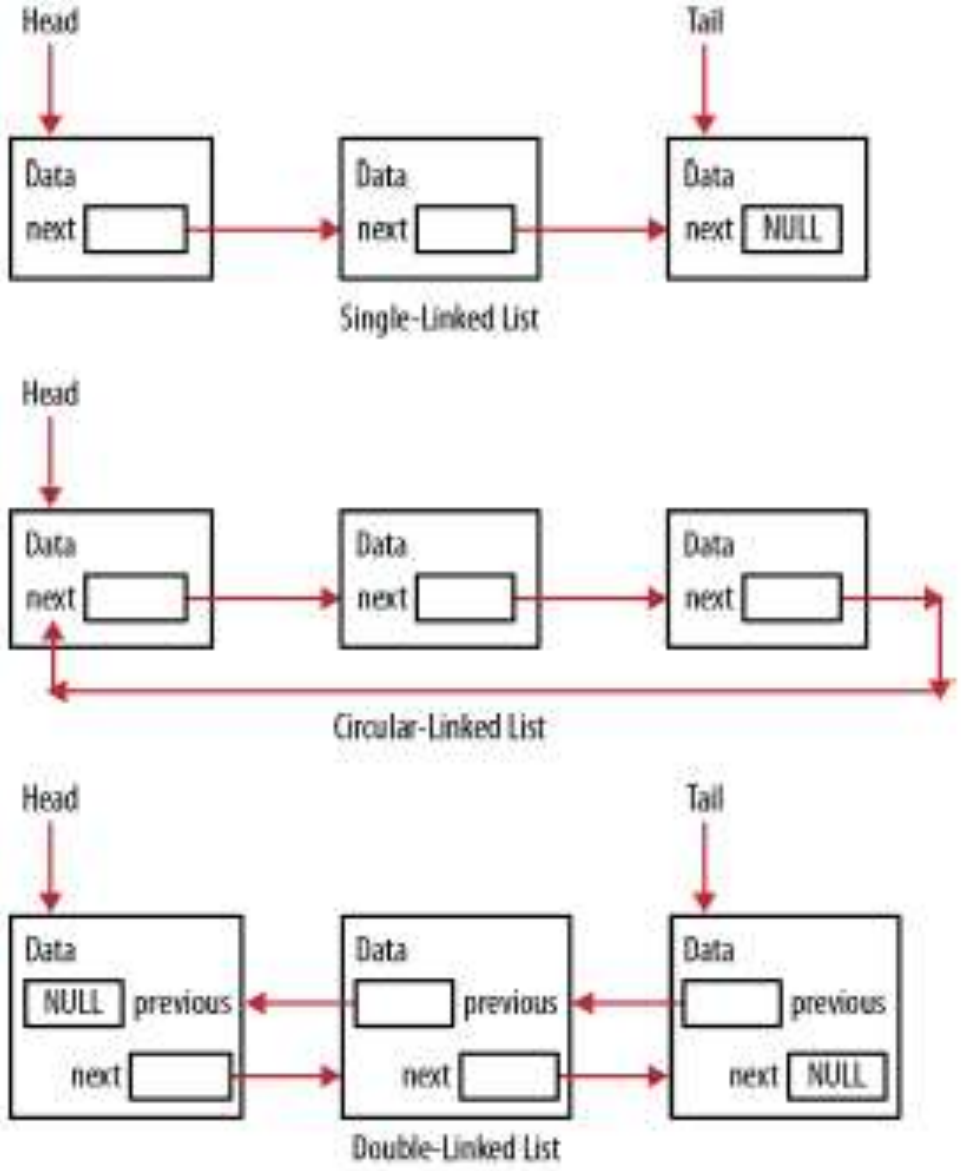
```
char* nomes[] = {“Cat", “Bronn", “Arya", “Mormon", "arya"};
sort(nomes,5,compare);
mostraNomes(nomes,5);
sort(nomes,5,compareCBaixa);
mostraNomes(nomes,5);
```

Atividades

- Implementar
  - Lista ligada
  - Fila
  - Pilha
  - Árvore

Atividades

- Implementar



Atividades

```
typedef void(*MOSTRA)(void*);
typedef int(*COMPARA)(void*, void*);
```

```
typedef struct _no {
        void *data;
        struct _no *proximo;
} No;
```

- void inicializaLista(ListaLigada*)
- void insereCabeca(ListaLigada*, void*)
- void insereCalda(ListaLigada*, void*)
- void mostraListaLigada(ListaLigada*, MOSTRA)
- No *getNo(ListaLigada*, COMPARA, void*)
- void remove(ListaLigada*, No*)

```
typedef struct _empregado{
        char nome[32];
        unsigned char idade;
} Empregado;
```

```
int comparaEmpregado(Empregado *e1, Empregado *e2) {
    return strcmp(e1->nome, e2->nome);
}
void mostraEmpregado(Empregado *empregado) {
    printf("%s\t%d\n", empregado->nome, empregado->idade);
}
```

Atividades

- Fila
- void inicializaFila(Fila*)
- void enfila(Fila*, void*)
- void *desenfila(Fila*)
-

Atividades

- Pilha
- void inicializaPilha(Pilha*)
- void empilha(Pilha*, void*)
- void *desempilha(Pilha*)
-

Referências

- Reese, R. Understanding and Using C pointers. 2013.