

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus Campina Grande

Curso Engenharia da Computação
Programação e Estruturas de Dados

Linguagem C: Estruturas e Alocação Dinâmica

Copyright©2018
Prof. César Rocha
cesarocha@ifpb.edu.br /

- Explorar os conceitos fundamentais acerca da **alocação dinâmica** de memória em C
 - Quando e como alocar memória, principais funções e bibliotecas da linguagem C, como acessar áreas alocadas, boas práticas de programação, quando e como liberar memória.
- Apresentar os mecanismos fundamentais da linguagem C para a **estruturação** de tipos
 - Por que estruturar código, como acessar os campos de uma estrutura, uso de typedef, ponteiros para estruturas, passando estruturas para funções, boas práticas e alocação dinâmica de estruturas

Parte I - Alocação Dinâmica de Memória

Motivação

- Até aqui, na declaração de um vetor, foi preciso dimensioná-lo (seja diretamente ou seja $v[n]$)
 - Isto nos obrigava a saber, de antemão, o número máximo de elementos no vetor durante a codificação
- Em alguns cenários, este pré-dimensionamento de memória é um **fator muito limitante**.
 - Imagine uma função que possa receber um vetor (de tamanho desconhecido) e tenha de criar um outro vetor de mesmo conteúdo (um clone) do original
 - Devo dimensionar o vetor clone com um número absurdamente alto ou ser modesto (e não prever vetores de tamanho maiores) ?

- Felizmente, C oferece meios de requisitar espaços de memória em tempo de execução
 - Dizemos que é possível **alocar memória dinamicamente**
- Assim, teremos programas que utilizam recursos de memória de maneira mais **versátil**
 - Só utilizam a memória quando necessário
 - Se adaptam aos recursos disponíveis da máquina
- Todas as operações de acesso e alteração dos dados alocados dinamicamente são feitas a partir dos **apontadores**

- Em C, basicamente, há três formas de se alocar memória, conforme a seguir:
 - ① **Estaticamente**: uso de variáveis **globais** e variáveis **locais estáticas** declaradas dentro de funções. Estas são liberadas apenas com o término do programa
 - ② **Implicitamente**: uso de variáveis **locais**. O espaço existe apenas enquanto a função que declarou a variável está sendo executada
 - ③ **Dinamicamente**: em tempo de execução, pode-se alocar um espaço de um determinado tamanho desejado. Este permanece reservado até que explicitamente seja liberado pelo programa(dor).

Graficamente

Alocação Esquemática da Memória

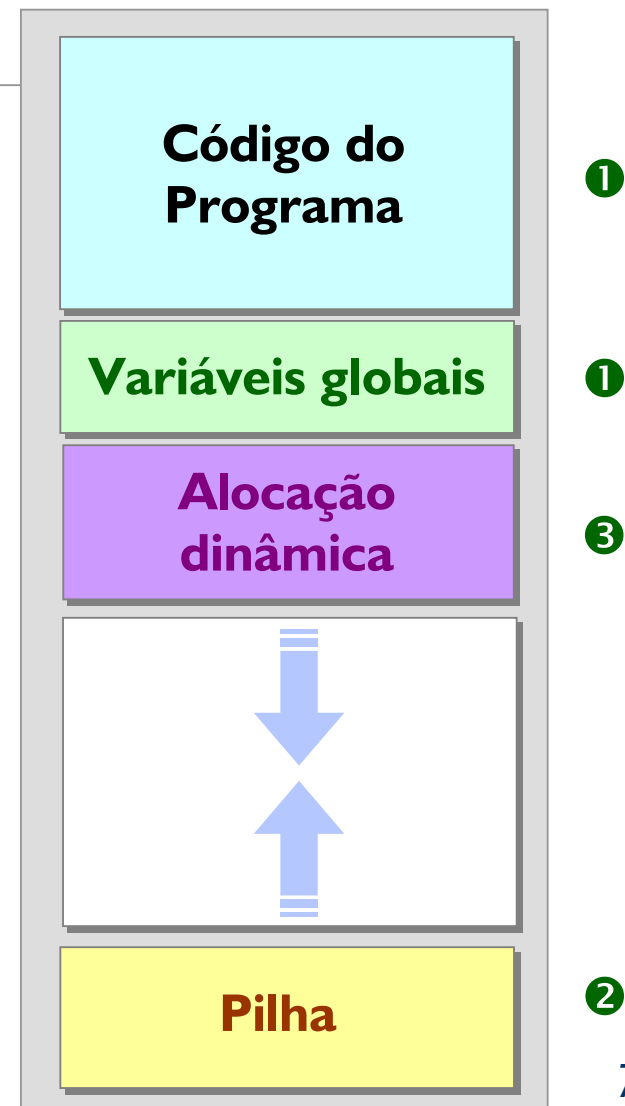
```
include <stdio.h>
int main( int argc, i
char *str;
printf("%s de cliente
// Numerical results
printf("Digite o pr
```

O sistema operacional aloca o código de máquina na memória.

Ele reserva também os espaços necessários para armazenar as variáveis globais (e estáticas) existentes no programa.

Esta área livre de memória é requisitada não somente pelas chamadas de função na pilha, mas também pelas alocações em tempo de execução.

O restante da memória livre é utilizado pelas variáveis locais (na pilha) e pelos espaços alocados dinamicamente.



malloc()

- **malloc()** é a principal função de alocação dinâmica de memória presente na **stdlib.h**
 - Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

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

- Se o espaço de memória livre for menor que o espaço requisitado dinamicamente, **malloc()** retorna um apontador nulo (NULL)
- O apontador também será utilizado quando se for fazer a liberação da área de memória alocada

malloc()

- *Por questões de portabilidade entre compiladores, utilize o operador **sizeof()** em vez de atribuir manualmente o tamanho em bytes do tipo*
- *Devemos lembrar que a função malloc é usada para alocar espaço de qualquer tipo, daí o **cast** explícito*

*// Considere o código abaixo na alocação de um vetor de inteiros
// de 4 células*

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

```
int main(void) {
```

```
    int *v;
```

```
    v = (int*) malloc ( 4 * sizeof( int ) );
```

```
    v[0] = 23; ...
```

```
}
```

Graficamente

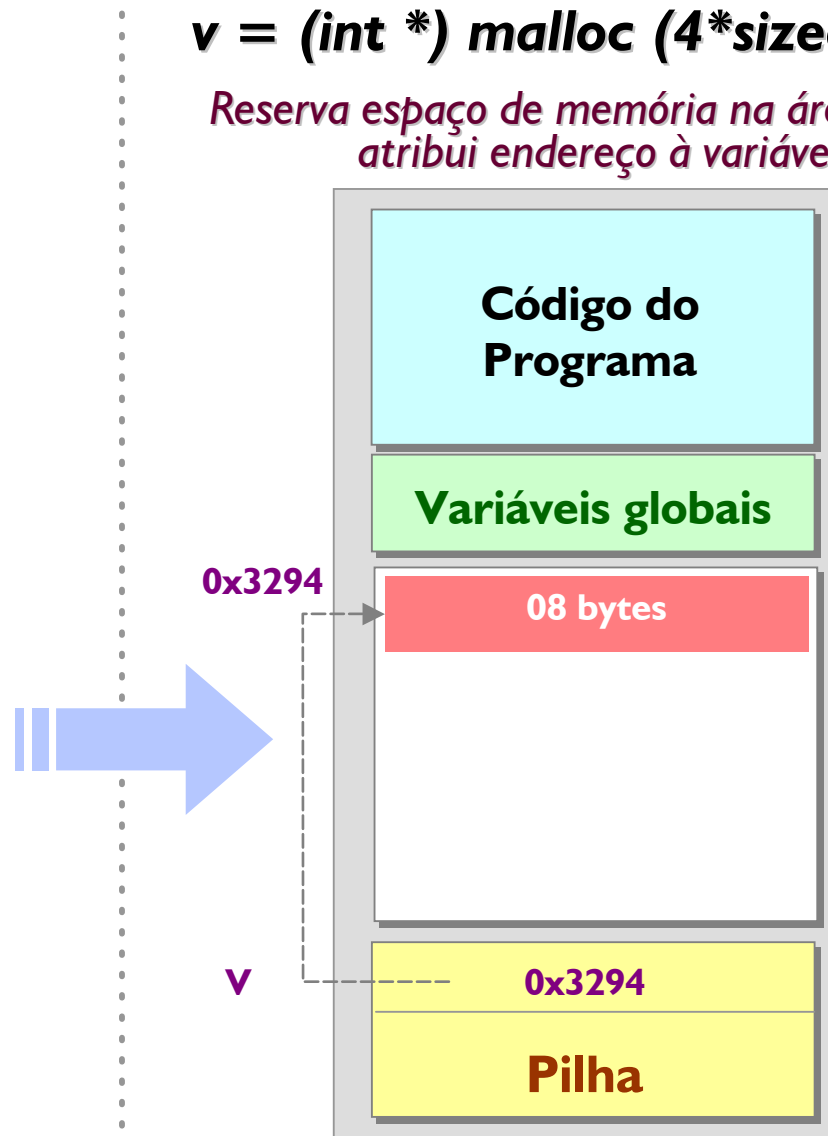
`int *v;`

Abre-se espaço na pilha para o ponteiro (variável local)



`v = (int *) malloc (4*sizeof(int))`

Reserva espaço de memória na área livre e atribui endereço à variável



Tratamento de erros

- **É fundamental testar** se a alocação teve ou não sucesso **antes de usar** o apontador retornado

```
// Considere o código abaixo na alocação de um vetor de inteiros  
// de tamanho 4
```

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

```
int main(void) {
```

```
    int *v;
```

```
    v = (int*) malloc ( 4 * sizeof( int ) );
```

```
    if (v==NULL) {
```

```
        puts( "Memória insuficiente.\n" );
```

```
        return(1); // aborta o programa e retorna 1 para o SO
```

```
    }
```

```
    v[0] = 23; ...
```

```
}
```

- *Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()***
 - *Esta função recebe como parâmetro o ponteiro contendo o endereço da memória a ser liberada.*

```
void free( void *pt );
```

- *Só podemos passar para esta função um endereço de memória que tenha sido alocado dinamicamente.*
- *Devemos lembrar ainda que não podemos acessar o espaço na memória depois que o liberamos.*
- *O uso de um apontador inválido passado como parâmetro para free pode derrubar o sistema.*

- Diferente de outras linguagens, é *tarefa do programador* liberar memória com free()

```
// Considere o código abaixo na alocação de um vetor de inteiros  
// de tamanho 4
```

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

```
int main(void) {
```

```
    int *v;
```

```
    v = (int*) malloc ( 4 * sizeof( int ) );
```

```
    if (v==NULL) {
```

```
        puts( "Memória insuficiente.\n" );
```

```
        return(1); // aborta o programa e retorna 1 para o SO
```

```
    }
```

```
    v[0] = 23; ...
```

```
    free( v );
```

```
}
```

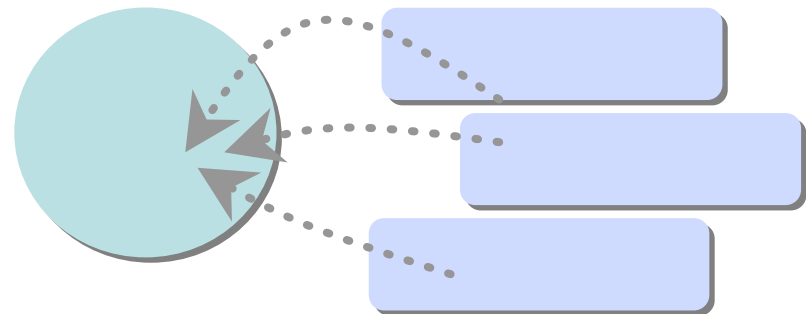
- *Utilizando-se de alocação dinâmica de memória, implemente uma função que receba como parâmetro uma cadeia de caracteres digitada pelo usuário e retorne uma cópia desta cadeia alocada dinamicamente.*
- *O protótipo desta função pode ser dado por:*
char* duplica (char* s);
- *Imprima a cópia ao final da duplicação*
- *A função que chama **duplica** ficará responsável por liberar o espaço alocado.*

Parte 2 – Estruturas

Motivação

- Vimos que C suporta tipos básicos (char, int, float, double, ...) vetores e ponteiros para estes tipos
 - Muitas vezes, na modelagem de uma entidade mais complexa do mundo real para o computacional, o programador C realiza esta tarefa *apenas reunindo vários valores de tipos mais simples*.
 - Ex: entidade livro num sistema de biblioteca

```
/* livro 1 */  
int cod; char titulo[40]; float preco;  
/* livro 2 */  
int cod2; char titulo2[40]; float preco2;
```



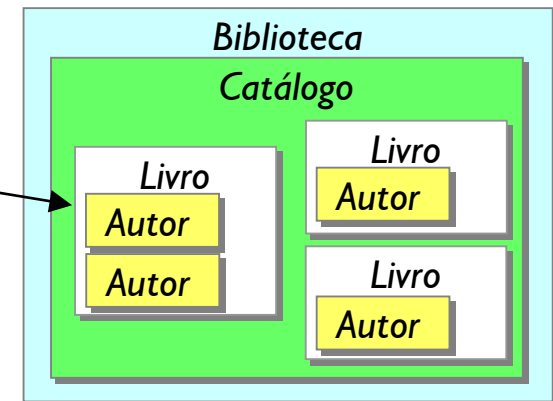
Deste modo, os campos ficam dissociados e, no caso de vários livros, cabe ao programador não misturar campos de um livro com os de outro.

Motivação

- Para amenizar este problema, C oferece recursos para agruparmos dados
 - Potencializar a **coesão** e **semântica** da entidade
 - Programas mais fáceis e com **melhor legibilidade**.
 - A definição de uma estrutura forma um modelo para representação de um TAD (Tipo Abstrato de Dado)

```
/* estruturação */  
struct livro {  
    int cod;  
    char titulo[40];  
    float preco; ..  
};
```

Lógica procedural
agrupada em
estruturas pequenas




Na estrutura acima, podemos agrupar os dados básicos numa estrutura única e coesa. O programador C pode criar entidades mais próximas do mundo real.

- Uma estrutura em C serve basicamente para agrupar diversas variáveis dentro de um único contexto.
 - Coleção de variáveis são referenciadas por um nome
 - Os membros de uma estrutura são geralmente chamados de elementos ou **campos**

```
struct <identificador> {  
    tipo nomeVariavel;  
    tipo nomeVariavel;  
    ...  
} variaveis estrutura;
```

```
struct aluno {  
    int matricula;  
    char nome[40];  
    int idade;  
} aluno1, aluno2;
```



Onde: *<identificador>* ou *variáveis estrutura* podem ser omitidos, mas nunca ambos!

Acessando os campos

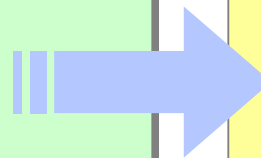
- Os elementos de uma estrutura podem ser acessados através do operador de acesso “ponto” (.)

```
#include <stdlib.h>
struct ponto { // definição to TAD ponto que possui coordenadas
    float x; // coordenada X do ponto
    float y; // coordenada Y do ponto
};
int main(void) {
    struct ponto p1; // declaração variável p1 do tipo ponto
    printf( "Digite as coordenadas do ponto( x, y ):" );
    scanf( "%f %f", &p1.x, &p1.y );
    printf( "O ponto fornecido foi: (%.2f,%.2f)\n", p1.x, p1.y );
    return 0;
}
```

- Visando *simplificar (ou abreviar)* nomes de tipos e para tratarmos tipos complexos, é recomendado o uso de **typedef**

- Por exemplo:

```
struct ponto {  
    float x;  
    float y;  
}  
struct ponto pl;
```



```
typedef struct {  
    float x;  
    float y;  
} Ponto; // novo tipo  
Ponto pl;
```

- Vale lembrar que typedef permite nomear quaisquer outros tipos de dados (float, int, ...)

Ponteiros para estruturas

- Da mesma forma que podemos declarar um ponteiro para uma variável comum, podemos armazenar o endereço de uma estrutura
 - Muito útil em *passagem de estruturas para funções* (veremos mais adiante)
- É possível acessar os campos dessa estrutura indiretamente através deste ponteiro
 - Operador **seta ->** é utilizado para acessar os elementos de uma estrutura que foi passada como ponteiro;

Ponteiros para estruturas

■ Exemplo:

```
#include <stdlib.h>
typedef struct { // definição TAD ponto com coordenadas
    float x; // coordenada X do ponto
    float y; // coordenada Y do ponto
} Ponto;
int main(void) {
    Ponto p1; // declaração variável p1 do tipo ponto
    Ponto *pt; // declaração do ponteiro para estrutura
    p1.x = 5; p1.y = 4; // acesso as variáveis via operador '.'
    p2 = &p1; // faz pt apontar para endereço da estrutura
    p2->x = 10; p2->y = 20 // acesso aos campos via ->
    printf( "Coordenadas de p1: (%.2f,%.2f)\n", p1.x, p1.y );
    return 0;
}
```

- Em síntese, se temos uma **variável** estrutura e queremos acessar seus campos, usamos o operador de acesso **ponto**
- Se temos uma variável **ponteiro** para estrutura, usamos o operador de acesso **seta**
- Finalmente e, seguindo o raciocínio, se temos uma **variável** ou um **ponteiro** e queremos acessar o **endereço** de um campo, fazemos **&**
 - Ex: **scanf("%f %f", &var.x, &var.y);**
scanf("%f %f", &pt->x, &pt->y);

Passando estruturas para funções

- *Vimos, no módulo anterior, que a passagem por parâmetros em C se dá fundamentalmente por valor*
 - *Não era possível alterar a variável original e sim apenas o parâmetro local, dentro da função*
 - *Era necessário utilizar o operador & quando queríamos passar não uma cópia, mas o endereço da variável*
- *Porém, no caso de estruturas, considere a função:*

```
void captura (struct ponto p) {  
    puts( "Digite as coordenadas (x y): " );  
    scanf("%f %f", &p.x, &p.y);  
}
```

Preste atenção nos pontos críticos!

Passando estruturas para funções

- *Existem alguns pontos críticos a serem considerados:*
 - *Copiar uma estrutura inteira para a pilha pode ser uma operação onerosa para o compilador. E se a estrutura tivesse 30 campos?*
 - *Não é possível alterar os campos da estrutura original*
- **Altamente recomendado:**
 - *É mais conveniente passar apenas o ponteiro da estrutura, mesmo que não seja necessário alterar os valores dos elementos dentro da função*

○ *que deve ser feito para corrigir a função anterior?*

Passando estruturas para funções

■ Exemplo:

```
void captura (struct ponto* pt) {  
    printf("Digite as coordenadas do ponto(x y): ");  
    scanf("%f %f", &pt->x, &pt->y);  
}
```

```
void imprime (struct ponto* pt) {  
    printf("O ponto fornecido: (%.2f,%.2f)\n", pt->x, pt->y);  
}
```

// função principal

```
int main(void) {  
    struct ponto p;  
    captura(&p);  
    imprime(&p);  
    return 0;  
}
```

Alocação dinâmica de estruturas

- *Da mesma forma que fizemos com vetores, as estruturas podem ser alocadas dinamicamente:*

```
#include <stdlib.h>
typedef struct { // definição TAD ponto com coordenadas
    float x; // coordenada X do ponto
    float y; // coordenada Y do ponto
} Ponto;
int main(void) {
    Ponto *pt; // declaração do ponteiro para estrutura
    pt = ( Ponto* ) malloc ( sizeof ( Ponto ) );
    if (pt==NULL) {
        puts( "Memória insuficiente.\n" );
        return 1; // aborta o programa e retorna 1 para o SO
    }
    pt->x = 42; ...
    free(pt);
```

Exercícios

- *(Para profissionais!) Escreva uma função que tenha como valor de retorno a distância entre dois pontos.*
 - *O protótipo da função final pode ser dado por:*
**float distancia(struct Ponto *p1,
 struct Ponto *p2);**
 - *Primeiro, crie os dois pontos (duas estruturas) alocados dinamicamente em memória. Depois, calcule a distância entre os pontos com base na fórmula abaixo:*

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- *A função que chama **distancia** ficará responsável por imprimir o resultado e liberar o espaço alocado*

Para um bom aproveitamento:

- *Codifique os exemplos mostrados nestes slides e verifique pontos de dúvidas*
- *Resolva todas as questões da **lista de alocação dinâmica e estruturas***
- *Procure o professor ou monitor da disciplina e questione conceitos, listas, etc.*
- *Não deixe para codificar tudo e acumular assunto para a primeira avaliação.*
 - *Este é apenas um dos assuntos abordados na prova!*