MC-202 — Aula 1
Revisão: Ponteiros, Alocação Dinâmica e Tipo Abstrato de Dados

Lehilton Pedrosa

Instituto de Computação – Unicamp

Segundo Semestre de 2015

Roteiro

1. Ponteiros
2. Alocação dinâmica
3. Registros
4. Tipo Abstrato de Dados

Um problema pra esquentar

Problema

Dados um conjunto de pontos do plano, como calcular o centroide?

Um problema pra esquentar

Problema

Dados um conjunto de pontos do plano, como calcular o centroide?

```
#include <stdio.h>
int main() {
    float x[100], y[100], cx, cy;
    int i, n;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%f %f", &x[i], &y[i]);

    cx = cy = 0;
    for (i = 0; i < n; i++) {
        cx = cx + x[i]; cy = cy + y[i];
    }
    cx = cx / n; cy = cy / n;
    printf("%f %f\n", cx, cy);
    return 0;
}
```

Operadores de ponteiro

Relembrando os operadores de ponteiros

O que era mesmo o & antes da variável?

Operadores de ponteiro

Relembrando os operadores de ponteiros

O que era mesmo o & antes da variável?

- O operador & retorna o endereço de memória de uma variável
- O operador $*$ acessa o conteúdo do endereço indicado pelo ponteiro

Operadores de ponteiro

Relembrando os operadores de ponteiros

O que era mesmo o & antes da variável?

- O operador & retorna o endereço de memória de uma variável
- O operador * acessa o conteúdo do endereço indicado pelo ponteiro

```
int *endereco;
int variavel = 90;
endereco = &variavel;

printf("Variavel: %d \n", variavel);
printf("Variavel: %d \n", *endereco);

printf("Endereço: %p \n", endereco);
printf("Endereço: %p \n", &variavel);
```

Operadores de ponteiro

Relembrando os operadores de ponteiros

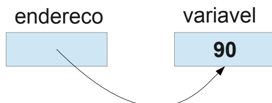
O que era mesmo o & antes da variável?

- O operador & retorna o endereço de memória de uma variável
- O operador $*$ acessa o conteúdo do endereço indicado pelo ponteiro

```
int *endereco;
int variavel = 90;
endereco = &variavel;

printf("Variavel: %d \n", variavel);
printf("Variavel: %d \n", *endereco);

printf("Endereço: %p \n", endereco);
printf("Endereço: %p \n", &variavel);
```



Alocação dinâmica

Problema

- Às vezes queremos queremos criar variáveis durante a execução.
- Mas por quê não declarar variáveis locais?

Alocação dinâmica

Problema

- Às vezes queremos queremos criar variáveis durante a execução.
- Mas por quê não declarar variáveis locais?
    1. Não sabemos quantas variáveis e quando declará-las
    2. Uma função pode ter que criar uma variável para outras funções usarem
    3. Queremos usar uma organização mais complexa da memória (*estrutura de dados*)

Pilha e Heap

Organização da memória

A memória de um programa é dividida em duas partes:

- **Pilha (Stack):** Onde são armazenadas as variáveis locais
- **Heap:** Onde são armazenadas as variáveis criadas pelo programador

Pilha e Heap

Organização da memória

A memória de um programa é dividida em duas partes:

- **Pilha (Stack):** Onde são armazenadas as variáveis locais
- **Heap:** Onde são armazenadas as variáveis criadas pelo programador

- **Variáveis locais:**
  - O compilador reserva um espaço na pilha
  - A variável é acessada por um nome bem definido
  - O espaço é liberado quando a função termina

Pilha e Heap

Organização da memória

A memória de um programa é dividida em duas partes:

- **Pilha (Stack):** Onde são armazenadas as variáveis locais
- **Heap:** Onde são armazenadas as variáveis criadas pelo programador

- **Variáveis locais:**
  - O compilador reserva um espaço na pilha
  - A variável é acessada por um nome bem definido
  - O espaço é liberado quando a função termina
- **Variáveis dinâmicas:**
  - O programador reserva um número de bytes no heap com **malloc**
  - Devemos guardar o endereço da variável com um ponteiro
  - O espaço deve ser liberado usando **free**

Exemplo

Criando uma variável inteira dinamicamente

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

int main() {
    int *ponteiro;

    ponteiro = malloc(sizeof(int));
    if (ponteiro == NULL) {
        printf("Não há mais memória!\n");
        exit(1);
    }

    *ponteiro = 13;
    printf("Endereco %p com valor %d.\n", ponteiro, *ponteiro);

    free(ponteiro);
    return 0;
}
```

Regras da alocação dinâmica

Regras:

- Devemos incluir a biblioteca `stdlib.h`
- O tamanho gasto por um tipo pode ser obtido com `sizeof`
- Devemos informar o tamanho a ser reservado para `malloc`
- Devemos verificar se acabou a memória comparando com `NULL`
- Devemos sempre liberar a memória após a utilização com `free`

Ponteiros e vetores

- Se escrevemos o nome de um vetor, obtemos um ponteiro para ele
- Podemo usar ponteiros como se fossem vetores

Ponteiros e vetores

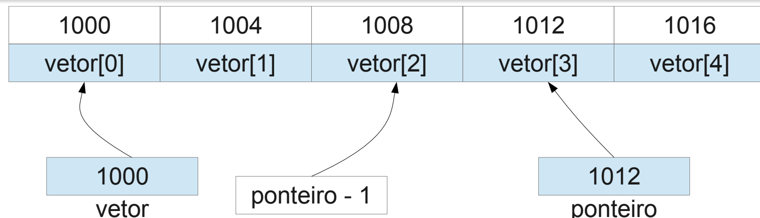
- Se escrevemos o nome de um vetor, obtemos um ponteiro para ele
- Podemo usar ponteiros como se fossem vetores

Lendo quantidade variável de notas

```
int main() {
    float *notas; // será usado como um vetor!
    int i, n;
    scanf("%d", &n);
    notas = malloc(n * sizeof(float));
    for (i = 0; i < n; i++)
        scanf("%d", &notas[i]);
    // ...
    free(notas);
    return 0;
}
```

Aritmética de ponteiros

- Vetores são ponteiros constantes: não podem ser alterados
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento




```
int vetor[5] = {1, 2, 3, 4, 5};
int *ponteiro;
ponteiro = vetor + 2;
ponteiro++;
printf("%d %d %d", *vetor, *(ponteiro - 1), *ponteiro)
```

Exemplo: centroide com malloc

Exemplo: centroide com malloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *x, *y, cx, cy;
    int i, n;

    scanf("%d", &n);
    x = malloc(n*sizeof(float));
    y = malloc(n*sizeof(float));
    if (x == NULL || y == NULL) {
        printf("Não há mais memória\n");
        exit(1);
    }
    for (i = 0; i < n; i++)
        scanf("%f %f", &x[i], &y[i]);
    ...
    printf("%f %f\n", cx, cy);

    free(x); free(y);
    return 0;
}
```

Registro — Definição

Não seria melhor se juntássemos as coordenadas de cada ponto?

Registro — Definição

Não seria melhor se juntássemos as coordenadas de cada ponto?

Registro

Registro é uma coleção de variáveis relacionadas de *vários* tipos, organizadas em uma única estrutura e referenciadas por um nome comum.

Registro — Definição

Não seria melhor se juntássemos as coordenadas de cada ponto?

Registro

Registro é uma coleção de variáveis relacionadas de *vários* tipos, organizadas em uma única estrutura e referenciadas por um nome comum.

Características

- Cada variável é chamada de membro do registro

Registro — Definição

Não seria melhor se juntássemos as coordenadas de cada ponto?

Registro

Registro é uma coleção de variáveis relacionadas de *vários* tipos, organizadas em uma única estrutura e referenciadas por um nome comum.

Características

- Cada variável é chamada de membro do registro
- Cada membro é acessado per um nome na estrutura

Registro — Definição

Não seria melhor se juntássemos as coordenadas de cada ponto?

Registro

Registro é uma coleção de variáveis relacionadas de *vários* tipos, organizadas em uma única estrutura e referenciadas por um nome comum.

Características

- Cada variável é chamada de membro do registro
- Cada membro é acessado per um nome na estrutura
- Cada estrutura define um novo tipo, com as mesmas características de um tipo padrão da linguagem

Declaração de estruturas e registros

Declarando uma **estrutura** com *n* membros

```
struct identificador {
     tipo1 membro1;
     tipo2 membro2;
     ...
     tipoN membroN;
};
```

Declaração de estruturas e registros

Declarando uma **estrutura** com *n* membros

```
struct identificador {
     tipo1 membro1;
     tipo2 membro2;
      ...
     tipoN membroN;
};
```

Declarando **um registro**

```
struct identificador nome_registro;
```

Declaração de estruturas e registros

Declarando uma **estrutura** com *n* membros

```
struct identificador {
     tipo1 membro1;
     tipo2 membro2;
      ...
     tipoN membroN;
};
```

Declarando **um registro**

```
struct identificador nome_registro;
```

Em C:

- Declaramos um tipo de uma estrutura apenas uma vez
- Podemos declarar vários registros da mesma estrutura

Exemplo de estrutura

Ficha de dados cadastrais de um aluno

```
struct data {
  int dia;
  int mes;
  int ano;
};

struct ficha_aluno {
  int ra;
  int telefone;
  char nome[30];
  char endereco[100];
  struct data nascimento;
};
```

Exemplo de estrutura

Ficha de dados cadastrais de um aluno

```
struct data {
  int dia;
  int mes;
  int ano;
};

struct ficha_aluno {
  int ra;
  int telefone;
  char nome[30];
  char endereco[100];
  struct data nascimento;
};
```

- Temos uma estrutura aninhada!

Usando um registro

Acessando um membro do registro

`registro.`*membro*

`ponteiro_registro->`*membro*

Usando um registro

Acessando um membro do registro

`registro`.*membro*

`ponteiro_registro`->*membro*

Imprimindo o nome de um aluno

```
struct ficha_aluno aluno;
struct ficha_aluno *ponteiro_aluno;
...
printf("Aluno: %s\n", aluno.nome);
printf("Outro aluno: %s\n", ponteiro_aluno->nome);
```

Usando um registro

Acessando um membro do registro

`registro.`*membro*

`ponteiro_registro->`*membro*

Imprimindo o nome de um aluno

```
struct ficha_aluno aluno;
struct ficha_aluno *ponteiro_aluno;
...
printf("Aluno: %s\n", aluno.nome);
printf("Outro aluno: %s\n", ponteiro_aluno->nome);
```

Imprimindo o aniversário

```
struct ficha_aluno aluno;
...
printf("Aniversario: %d/%d\n", aluno.nascimento.dia,
                               aluno.nascimento.mes);
```

Exemplo com Registros

Problema

Vamos reescrever o programa do centroide usando registros. Mas agora:

- os pontos são multidimensionais
- a dimensão é lida do teclado no início
- vamos ignorar pontos repetidos

Pare! vamos refletir

O que fizemos até agora?

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita
  - o compilador esconde os detalhes!

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita
  - o compilador esconde os detalhes!
  - o tipo `float` nada mais é do que uma "abstração" de um registrador

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita
  - o compilador esconde os detalhes!
  - o tipo `float` nada mais é do que uma "abstração" de um registrador
- mas quando somamos 2 pontos? ou comparamos 2 pontos?
  - nos preocupamos com os detalhes

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita
  - o compilador esconde os detalhes!
  - o tipo `float` nada mais é do que uma "abstração" de um registrador
- mas quando somamos 2 pontos? ou comparamos 2 pontos?
  - nos preocupamos com os detalhes
  - será que também podemos abstrair um ponto?

Pare! vamos refletir

O que fizemos até agora?

- quando somamos 2 variáveis `float`?
  - não nos preocupamos como a operação é feita
  - o compilador esconde os detalhes!
  - o tipo `float` nada mais é do que uma "abstração" de um registrador
- mas quando somamos 2 pontos? ou comparamos 2 pontos?
  - nos preocupamos com os detalhes
  - será que também podemos abstrair um ponto?
  - **Sim! com registro, funções e um pouco de cuidado**

Tipo Abstrato de Dados — Definição

Tipo Abstrato de Dados

Tipo Abstrato de Dados (TAD) é um conjunto de valores associado a um conjunto de operações permitidas nesses dados.

Tipo Abstrato de Dados — Definição

Tipo Abstrato de Dados

Tipo Abstrato de Dados (TAD) é um conjunto de valores associado a um conjunto de operações permitidas nesses dados.

- **Interface** é o conjunto de operações de uma TAD. Ela consiste dos nomes e demais convenções usadas para executar cada operação.
- **Implementação** é o conjunto de algoritmos que realiza as operações. A implementação é o único "lugar" que uma variável ou dado é acessada diretamente.
- **Cliente** é o código que utiliza/chama uma operação. O cliente nunca acessa a variável diretamente.

Tipo Abstrato de Dados — Definição

Tipo Abstrato de Dados

Tipo Abstrato de Dados (TAD) é um conjunto de valores associado a um conjunto de operações permitidas nesses dados.

- **Interface** é o conjunto de operações de uma TAD. Ela consiste dos nomes e demais convenções usadas para executar cada operação.
- **Implementação** é o conjunto de algoritmos que realiza as operações. A implementação é o único “lugar” que uma variável ou dado é acessada diretamente.
- **Cliente** é o código que utiliza/chama uma operação. O cliente nunca acessa a variável diretamente.

Em C

- um TAD é declarado como um registro (`struct`)
- a interface é um conjunto de protótipos de funções que recebe o registro.

Um exemplo concreto: retângulo

É uma boa prática criar 2 arquivos: a **interface** e a implementação

Interface: retangulo.h

```
typedef struct {
    char cor[10];
    float largura, altura;
} Retangulo;

// aloca memória e inicializa
Retangulo *criar_retangulo();

float altura_retangulo(Retangulo *ret);
float largura_retangulo(Retangulo *ret);
float area_retangulo(Retangulo *ret);
void ler_retangulo(Retangulo *ret);
void girar_retangulo(Retangulo *ret);

// finaliza e libera memória
void destruir_retangulo(Retangulo *ret);
```

Um exemplo concreto: retângulo

É uma boa prática criar 2 arquivos: a interface e a **implementação**

Implementação: retangulo.c

```
Retangulo *criar_retangulo() {
    Retangulo *r = malloc(sizeof(Retangulo));
    if (r == NULL) {
        printf("Faltou memória\n"); exit(1);
    }
    r->largura = r->altura = 0; // retângulo vazio
    return r;
}

float area_retangulo(Retangulo *r) {
    return r->largura * r->altura;
}
...
void destruir_retangulo(Retangulo *r) {
    free(r);
}
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

Cliente: um caso de teste

```c
#include <stdio.h>
#include "retangulo.h"
int main() {
    float area1, area2;
    Retangulo *r;
    r = criar_retangulo();
    ler_retangulo(r);

    area1 = area_retangulo(r);
    girar_retangulo(r);
    area2 = area_retangulo(r);

    if (area1 != area2) {
        printf("A implementação está incorreta!\n");
    }
    destruir_retangulo(r);
    return 0;
}
```

Exercício 1

1. Quando um programa inicia, ele é carregado pelo sistema operacional na memória. Pesquise como essa memória é organizada e responda:
   - Quais são as principais partes da memória?
   - Em que partes são armazenadas: variáveis locais, variáveis estáticas, strings constantes, números constantes no programa, variáveis alocadas com malloc, comandos?
2. Reflita e responda:
   - Qual a diferença entre passagem por valor e referência?
   - Quando é vantajoso em passar registros (struct) por referências?
   - E quando é melhor usar passagem por valores?
3. Complete a implementação e interface dos TADs retângulo e ponto vistos em sala.
4. **Desafio/extra:** Crie um TAD polígono que inclua as operações:
   - é polígono simples (não contém cruzamentos);
   - área do polígono (assuma que o polígono é convexo)

Exercício 2

Matriz linearizada

Uma outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha ou por uma coluna.

Escreva um programa que calcule o centroide de um conjuntos de pontos e calcule a soma das distâncias de cada ponto para o centroide. Serão lidos em ordem a dimensão dos pontos, o número de pontos e as coordenadas em ordem de cada ponto.

Você deve utilizar alocação dinâmica de memória e uma matriz para armazenar os pontos. Assim, a matriz deverá ser representada na memória como um vetor. Como isso pode ser feito?