TAD - Tipo Abstrato de Dados

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2025

Tipos Abstratos de Dados (TADs)

- Um Tipo Abstrato de Dados (TAD) é uma especificação de um **conjunto de dados** e das **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações

Tipos Abstratos de Dados (TADs)

- Um Tipo Abstrato de Dados (TAD) é uma especificação de um **conjunto de dados** e das **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações
- A ideia central é encapsular (esconder) de quem usa um determinado tipo de dado a forma concreta com que ele foi implementado.
- Os usuários do TAD só têm acesso a algumas operações disponibilizadas sobre esses dados. Eles não têm acesso a detalhes de implementação.

Tipos Abstratos de Dados (TADs)

- Um Tipo Abstrato de Dados (TAD) é uma especificação de um **conjunto de dados** e das **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações
- A ideia central é encapsular (esconder) de quem usa um determinado tipo de dado a forma concreta com que ele foi implementado.
- Os usuários do TAD só têm acesso a algumas operações disponibilizadas sobre esses dados. Eles não têm acesso a detalhes de implementação.
  - Comportamento semelhante acontece quando usamos as bibliotecas padrão do C: stdio.h, string.h, stdlib.h, math.h, etc.

Características de um TAD

Um tipo abstrato de dados possui duas partes principais:

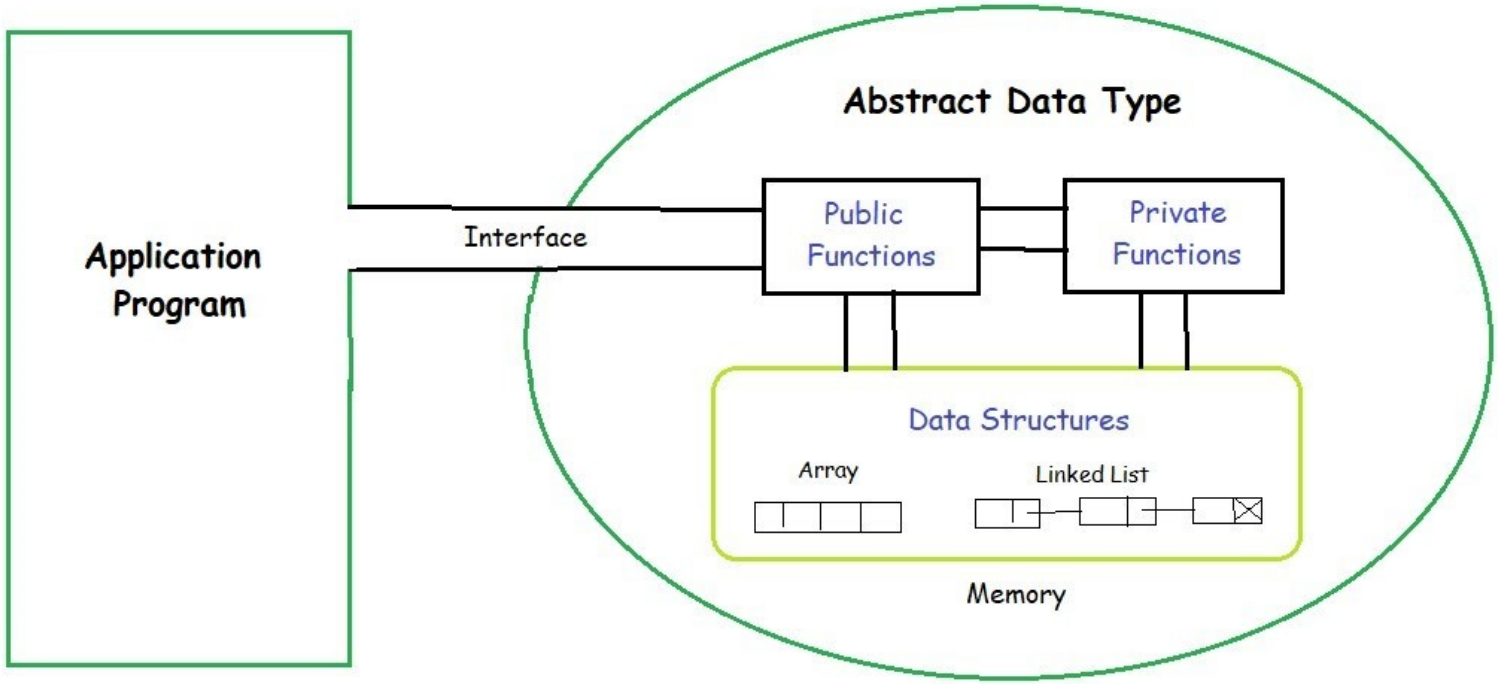
- **Interface:** define o comportamento do TAD, como as operações do tipo podem ser executadas, mas não como essas operações são implementadas.
  - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.

Características de um TAD

Um tipo abstrato de dados possui duas partes principais:

- **Interface:** define o comportamento do TAD, como as operações do tipo podem ser executadas, mas não como essas operações são implementadas.
  - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.
- **Representação concreta:** é a implementação em si, que nos diz:
  - como um TAD foi implementado.
  - como seus dados são colocados dentro do computador.
  - como estes dados são manipulados por suas operações (funções).

Tipos Abstratos de Dados (TADs)



- É chamado de "abstrato" porque fornece uma visão independente da implementação.
- O processo de fornecer apenas o essencial e ocultar os detalhes é conhecido como abstração.

Como implementar um TAD?

- A chave para se conseguir implementar tipos abstratos de dados é aplicar o conceito de Independência de Representação:
  - Um programa deveria ser projetado de forma que a representação de um tipo de dado possa ser modificada sem que isto interfira no restante do programa.

Como implementar um TAD?

- A chave para se conseguir implementar tipos abstratos de dados é aplicar o conceito de Independência de Representação:
  - Um programa deveria ser projetado de forma que a representação de um tipo de dado possa ser modificada sem que isto interfira no restante do programa.

- A aplicação deste conceito é melhor realizada através:
  - da modularização do programa e da definição de tipos opacos (em programação estruturada)
  - criação de classes (em programação orientada a objetos)

Como implementar um TAD?

- A chave para se conseguir implementar tipos abstratos de dados é aplicar o conceito de Independência de Representação:
  - Um programa deveria ser projetado de forma que a representação de um tipo de dado possa ser modificada sem que isto interfira no restante do programa.

- A aplicação deste conceito é melhor realizada através:
  - da modularização do programa e da definição de tipos opacos (em programação estruturada)
  - criação de classes (em programação orientada a objetos)

- Nesta disciplina, como estamos usando a linguagem C, vamos usar **módulos** e **tipos opacos** para implementar TADs.

Tipo opaco

Definição

Um tipo opaco em C é um tipo de dado cuja estrutura interna é escondida do usuário do módulo. Ou seja, o código que usa o tipo sabe que ele existe, mas não sabe nem pode acessar seus campos diretamente.

Exemplo de um tipo opaco — FILE

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Um exemplo de tipo opaco na linguagem C é o tipo FILE. Ele é uma struct que contém as informações sobre um arquivo necessárias para realizar as operações de entrada e saída sobre ele.

```
1  typedef struct {
2      int level;                  // nível do buffer
3      unsigned flags;             // flag de status do arquivo
4      char fd;                    // descritor do arquivo
5      unsigned char hold;         // retorna caractere sem buffer
6      int bside;                  // tamanho do buffer
7      unsigned char *buffer;      // buffer de dados
8      unsigned char *curp;        // ponteiro atualmente ativo
9      unsigned istemp;            // indicador de arquivo temporário
10     short token;                // usado para validação
11 }File;
```

Exemplo de um tipo opaco — FILE

- A única forma de trabalhar com arquivos em linguagem C é declarando um ponteiro de arquivo, da seguinte maneira:

```
FILE *ptr;
```

- A única maneira de acessar o conteúdo do ponteiro **FILE** é por meio das operações definidas em sua **interface** como, por exemplo:
  - fopen(), fclose(), fputc(), fgetc(), feof(), etc.

Por que usar tipos opacos?

Usar tipos opacos é uma boa prática de encapsulamento em C.
Algumas vantagens são:

- **Ocultamento de implementação:** você pode mudar a estrutura interna sem quebrar o código do usuário.
- **Segurança:** o usuário não consegue modificar campos diretamente (evita inconsistências).
- **Interfaces limpas:** o usuário trabalha apenas com funções da API.
- **Compilação mais rápida:** mudanças internas no arquivo .c não forçam recompilação de todos os arquivos que incluem o header .h

Modularização

Modularizando o programa

- Quando trabalhamos com TAD, a convenção em linguagem C é prepararmos dois arquivos para implementar o TAD. Assim, podemos separar o "conceito" (a definição do tipo) de sua "implementação":

Modularizando o programa

- Quando trabalhamos com TAD, a convenção em linguagem C é prepararmos dois arquivos para implementar o TAD. Assim, podemos separar o "conceito" (a definição do tipo) de sua "implementação":
  - **arquivo de cabeçalho .h** — aqui são declarados os protótipos das funções visíveis para o usuário, tipos de ponteiros, e dados globalmente acessíveis. Aqui é definida a **interface** visível pelo usuário.

Modularizando o programa

- Quando trabalhamos com TAD, a convenção em linguagem C é prepararmos dois arquivos para implementar o TAD. Assim, podemos separar o "conceito" (a definição do tipo) de sua "implementação":
  - **arquivo de cabeçalho .h** — aqui são declarados os protótipos das funções visíveis para o usuário, tipos de ponteiros, e dados globalmente acessíveis. Aqui é definida a **interface** visível pelo usuário.
  - **arquivo-fonte .c** — definição do tipo de dados que ficará oculto do usuário do TAD e implementação das suas funções. Aqui é definido tudo o que ficará oculto do usuário.

O que é um módulo?

Definição

Em C, um **módulo** é um conjunto de tipos de dados e de funções com um propósito único e bem definido e que pode ser compilado separadamente do restante do programa.

- Um módulo pode ser facilmente reutilizado e modificado independente do programa do usuário.

O que é um módulo?

Definição

Em C, um **módulo** é um conjunto de tipos de dados e de funções com um propósito único e bem definido e que pode ser compilado separadamente do restante do programa.

- Um módulo pode ser facilmente reutilizado e modificado independente do programa do usuário.
- À medida que uma aplicação se torna maior, o uso de módulos se torna necessário. Isso ocorre porque o uso de um único arquivo causa uma série de problemas!

Etapas da Compilação em C

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

- O pré-processador executa diretivas iniciadas com #, como:

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

- O pré-processador executa diretivas iniciadas com #, como:
  - `#include` — insere o conteúdo de headers

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

- O pré-processador executa diretivas iniciadas com #, como:
  - `#include` — insere o conteúdo de headers
  - `#define` — substitui macros

Compilação em C — Etapa 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

- O pré-processador executa diretivas iniciadas com #, como:
  - `#include` — insere o conteúdo de headers
  - `#define` — substitui macros
  - `#ifdef`, `#endif`, `#if` — controle condicional de compilação

Compilação em C — Etapa 1

- **Etapa 1. Pré-processamento**
  - Entrada: código-fonte (.c)
  - Saída: código-fonte expandido (.i)

- O pré-processador executa diretivas iniciadas com #, como:
  - `#include` — insere o conteúdo de headers
  - `#define` — substitui macros
  - `#ifdef`, `#endif`, `#if` — controle condicional de compilação

- O comando abaixo gera o código-fonte expandido:
  `gcc -E main.c -o main.i`

Compilação em C — Etapa 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 2. Compilação** (análise e tradução para assembly)

Compilação em C — Etapa 2

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)

Compilação em C — Etapa 2

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

Compilação em C — Etapa 2

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

- Nesta etapa, o compilador faz:

Compilação em C — Etapa 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

- Nesta etapa, o compilador faz:
  - **Análise léxica e sintática:** detecta erros de linguagem

Compilação em C — Etapa 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

- Nesta etapa, o compilador faz:
  - **Análise léxica e sintática:** detecta erros de linguagem
  - **Otimizações:** melhora o desempenho do código

Compilação em C — Etapa 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

- Nesta etapa, o compilador faz:
  - **Análise léxica e sintática:** detecta erros de linguagem
  - **Otimizações:** melhora o desempenho do código
  - **Geração de código assembly**

Compilação em C — Etapa 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 2. Compilação** (análise e tradução para assembly)
  - Entrada: arquivo pré-processado (.i)
  - Saída: código assembly (.s)

- Nesta etapa, o compilador faz:
  - **Análise léxica e sintática:** detecta erros de linguagem
  - **Otimizações:** melhora o desempenho do código
  - **Geração de código assembly**

- Gera o assembly correspondente ao seu código C:
  `gcc -S main.c -o main.s`

Compilação em C — Etapa 3

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 3. Montagem**

Compilação em C — Etapa 3

- **Etapa 3. Montagem**
  - Entrada: Entrada: código assembly (.s)

Compilação em C — Etapa 3

- **Etapa 3. Montagem**
  - Entrada: Entrada: código assembly (.s)
  - Saída: código objeto (.o)

Compilação em C — Etapa 3

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 3. Montagem**
  - Entrada: Entrada: código assembly (.s)
  - Saída: código objeto (.o)

- O montador traduz instruções assembly em código de máquina (binário), criando um arquivo objeto relocável.

Compilação em C — Etapa 3

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 3. Montagem**
  - Entrada: Entrada: código assembly (.s)
  - Saída: código objeto (.o)

- O montador traduz instruções assembly em código de máquina (binário), criando um arquivo objeto relocável.

- Esse .o contém o código em formato binário, mas ainda não é executável — faltam referências a funções externas (printf, etc.)

Compilação em C — Etapa 4

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Etapa 4. Ligação (*linking*)**

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

- Nessa etapa, o ligador (*linker*):

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

- Nessa etapa, o ligador (*linker*):
  - Junta todos os módulos (.o) do projeto

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

- Nessa etapa, o ligador (*linker*):
  - Junta todos os módulos (.o) do projeto
  - Resolve símbolos externos (funções de outras unidades e da libc)

Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

- Nessa etapa, o ligador (*linker*):
  - Junta todos os módulos (.o) do projeto
  - Resolve símbolos externos (funções de outras unidades e da libc)
  - Produz o executável final

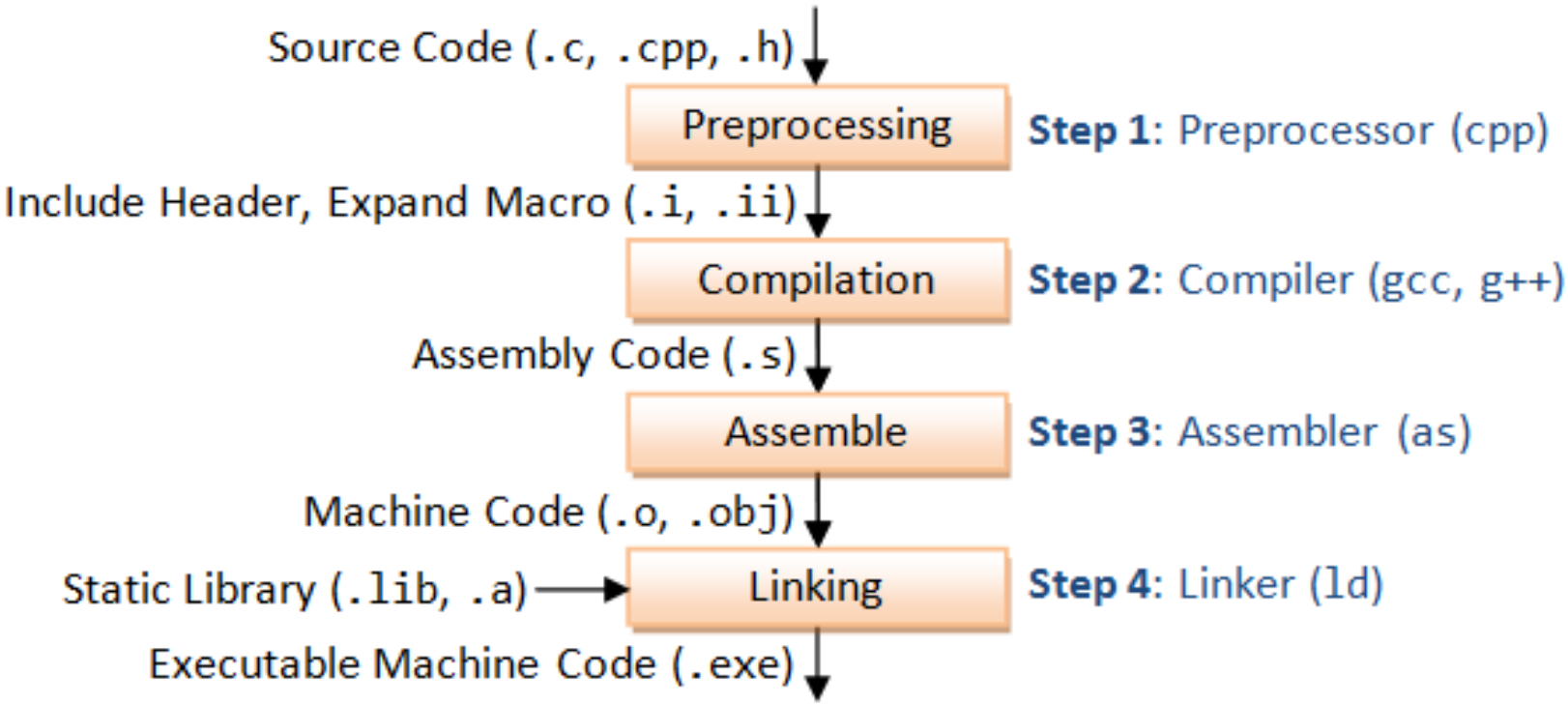
Compilação em C — Etapa 4

- **Etapa 4. Ligação (*linking*)**
  - Entrada: um ou mais arquivos objeto (.o) e bibliotecas
  - Saída: executável (a.out ou main)

- Nessa etapa, o ligador (*linker*):
  - Junta todos os módulos (.o) do projeto
  - Resolve símbolos externos (funções de outras unidades e da libc)
  - Produz o executável final

- O comando abaixo gera o binário completo, pronto para execução:
  `gcc main.o -o programa`

Visão geral das etapas



Compilação em C

Para ver cada etapa separadamente com o GCC:

```
gcc -E main.c -o main.i # Pré-processamento
gcc -S main.c -o main.s # Compilação
gcc -c main.c -o main.o # Montagem
gcc main.o -o main # Linkagem
```

Exercícios

TAD Point

- Criar de um TAD para representar um ponto (x, y) no espaço $\mathbb{R}^2$.

- Todo ponto no plano $\mathbb{R}^2$ possui duas coordenadas x e y. Estes são os dois atributos de um ponto.

- Consideramos as seguintes operações:
  - criar um ponto com coordenadas x e y
  - se for necessário, liberar a memória alocada por um ponto
  - devolver a coordenada x de um ponto
  - devolver a coordenada y de um ponto
  - atribuir novo valor à coordenada x do ponto
  - atribuir novo valor à coordenada y do ponto
  - calcular a distância entre dois pontos.

Exercício: Implemente um TAD chamado `Point` como um módulo, seguindo os requisitos listados acima.

Exercício — TAD Circle

- Criar um TAD para representar um círculo no $\mathbb{R}^2$.
- Implemente o TAD usando módulo e um tipo opaco `Circle`. Todo círculo pode ser definido a partir do seu **centro** e do seu **raio**.
- O módulo deve ter as seguintes funções:
  - `Circle* create_circle(double radius, const Point *center)`: cria um círculo cujo centro é um atributo do tipo `Point` e raio é um `double`.
  - `void setRadius(Circle *c, double r)`: atribui novo valor ao raio do círculo.
  - `void setCenter(Circle *c, double x, double y)`: atribui novo valor ao centro.
  - `void setCenter(const Point *p)`: muda o centro.
  - `double getRadius(const Circle *c)` obtém o raio.
  - `Point getCenter(const Circle *c,)`: obtém o centro.
  - `double area(const Circle *c,)`: retorna a área do círculo.
  - `bool contains(const Circle *c, const Point& p)`: verifica se o ponto p está dentro do círculo.

Exercício — TAD Matrix

- Criar um TAD para representar uma matriz com n linhas e m colunas. Os valores n e m são determinados no momento da criação da matriz.

- Implemente o TAD por meio de uma classe chamada `Matrix`.
  Esse TAD encapsula uma matriz com n linhas e m colunas sobre a qual podemos fazer as seguintes operações:

  - criar matriz alocada dinamicamente
  - destruir a matriz alocada dinamicamente
  - acessar valor na posição (i, j) da matriz
  - atribuir valor ao elemento na posição (i, j)
  - retornar o número de linhas da matriz
  - retornar o número de colunas da matriz
  - imprimir a matriz na tela do terminal
  - comparar a matriz com outra e decidir se são ou não iguais.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM