


# Tipos Abstratos de Dados I

23/10/2024

# Sugestões de leitura

- Estes slides apresentam exemplos disponíveis na Web
- Recomenda-se a leitura das seguintes páginas:
  - <https://www.geeksforgeeks.org/abstract-data-types/>
  - <https://www.edn.com/5-simple-steps-to-create-an-abstract-data-type-in-c/>

# Sumário

- Introdução: O que é um TAD ?
- Interface de um TAD vs Implementação de um TAD
- Invariantes da representação
- Análise de exemplos simples
- Convenções e organização em ficheiros
- Exercícios / Tarefas 

# Introdução

– O que é um TAD ?

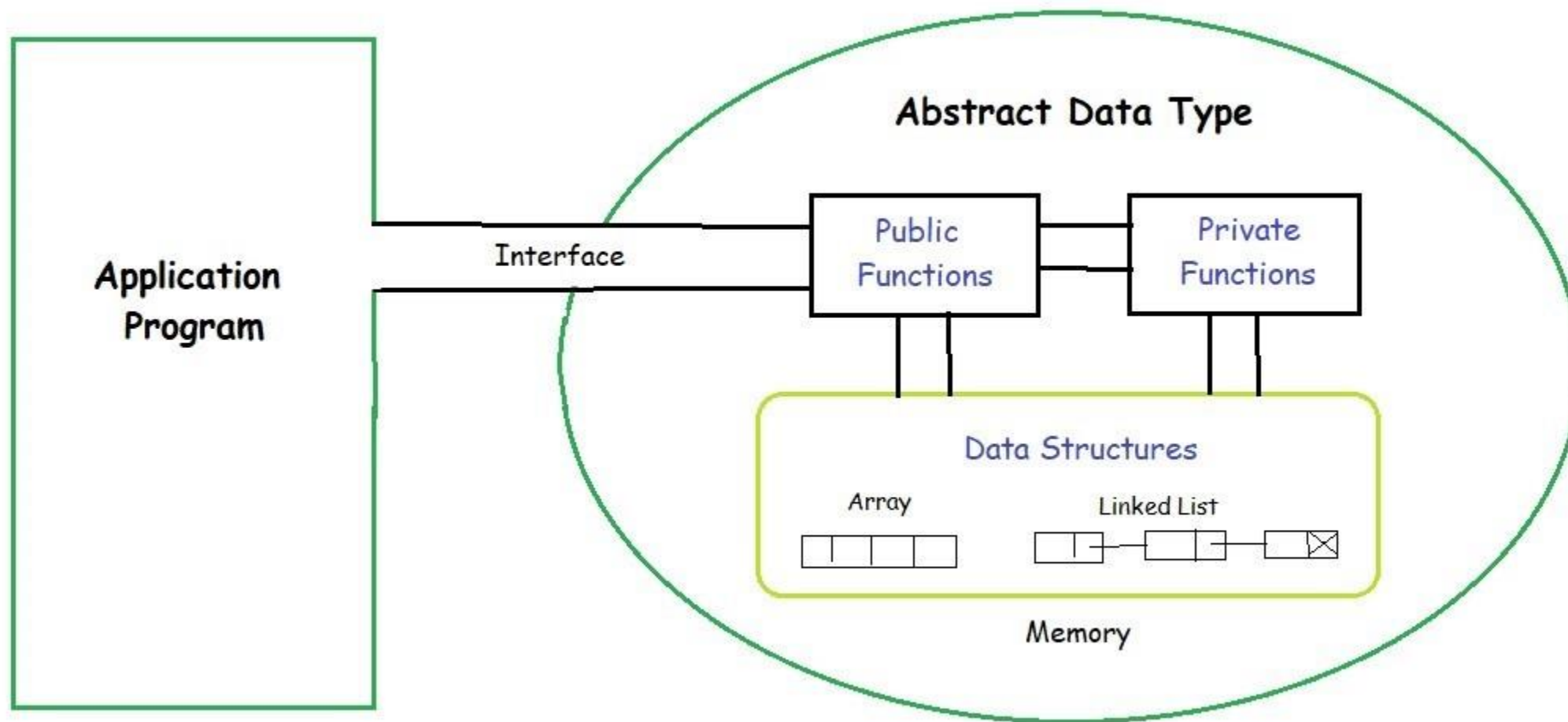
# Motivação

- A linguagem C **não** suporta o paradigma **OO**
- **MAS**, é possível usar alguns princípios de OO no desenvolvimento de código em C
- Uma estrutura de dados e as suas operações podem ser organizadas como um **Tipo Abstrato de Dados (TAD)**

# Tipo Abstrato de Dados (TAD)

- Tipo (i.e., classe) de instâncias
  - **Estado**: atributos
  - **Comportamento**: operações
- Define que **operações** são **possíveis**
  - **MAS** não como são implementadas !
- **NÃO** especifica como se organiza a informação em memória !
  - **Encapsulamento**
  - Representações / Estruturas de dados **alternativas**
- Tipo **ABSTRATO**: independente da implementação

# Tipo Abstrato de Dados (TAD)



[geeksforgeeks.org]

# Tipo Abstrato de Dados (TAD)

- Define uma **INTERFACE** entre o TAD e as aplicações que o usam
- **ENCAPSULA** os detalhes da **representação interna** das suas instâncias e da **implementação** das suas funcionalidades
  - Estão **ocultos** para os utilizadores do TAD !!
- **Detalhes** de representação / implementação **podem ser alterados** sem alterar a interface do TAD !
  - **Não** é necessário **alterar código que use o TAD** !!



# Como fazer ?

- Usar um **struct** para definir / armazenar os **atributos** de cada instância de um TAD
- As **funções** que operam sobre instâncias de um TAD são **definidas e implementadas** separadamente
- Vamos analisar exemplos simples...

# O TAD Triângulo

– 1ª versão

# Exemplo – O TAD Triângulo

- Que **operações** são possíveis sobre / com triângulos ?
- Qual é a **interface** do TAD ?
- Como **definir / representar** triângulos ?
- São necessárias / úteis **operações adicionais** / auxiliares ?

# O TAD Triângulo – 1ª versão – Atributos

```
// A triangle ADT.
```

```
struct Triangle {
```

```
    double a;
```

```
    double b;
```

```
    double c;
```

```
};
```

```
int main() {
```

```
    Triangle t1 = { 3, 4, 5 };
```

```
    Triangle t2 = { 2, 2, 2 };
```

```
}
```




- Comprimento de cada um dos lados






- Inicialização direta !
  - Obriga a conhecer detalhes da representação !!
  - NÃO FAZER ASSIM !!
  - Usar um construtor !


# O TAD Triângulo – 1ª versão – Operações



```
// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the perimeter of the given Triangle.
double Triangle_perimeter(const Triangle *tri) {
    return tri->a + tri->b + tri->c;
}
```



```
// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS: Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
    tri->a *= s;
    tri->b *= s;
    tri->c *= s;
}
```



# Convenções habituais

- Designação do TAD é um **prefixo** do nome de cada **função** da **interface**
- O **primeiro argumento** de cada função é um **ponteiro para a instância** com / sobre a qual opera
- Se essa instância **não é modificada**, declara-se o ponteiro como ponteiro para uma instância **constante**

# O TAD Triângulo – 1ª versão

- Implementação de **Triangle\_scale()** e **Triangle\_perimeter()** deverá estar encapsulada / oculta
- **Abstração**: suficiente conhecer as funcionalidades implementadas
- **MAS**, ao inicializar **t1** e **t2** há **acesso direto** à representação interna !!

# O TAD Triângulo – 1ª versão

- O que aconteceria, se fosse **alterada a representação** interna, mantendo os **3 atributos**, mas um deles com um **significado diferente** ?
- Melhor efetuar a **inicialização dos atributos** de cada instância usando uma **função construtora** !!
- E **assegurar a validade** de qualquer instância criada !!



# O TAD Triângulo – 1ª Versão – Construtor

```
// REQUIRES: tri points to a Triangle object ←
// MODIFIES: *tri
// EFFECTS:  Initializes the triangle with the given side lengths. ←
void Triangle_init(Triangle *tri, double a_in,
                  double b_in, double c_in) {
    ↑
    tri->a = a_in;
    tri->b = b_in; ←
    tri->c = c_in;
}

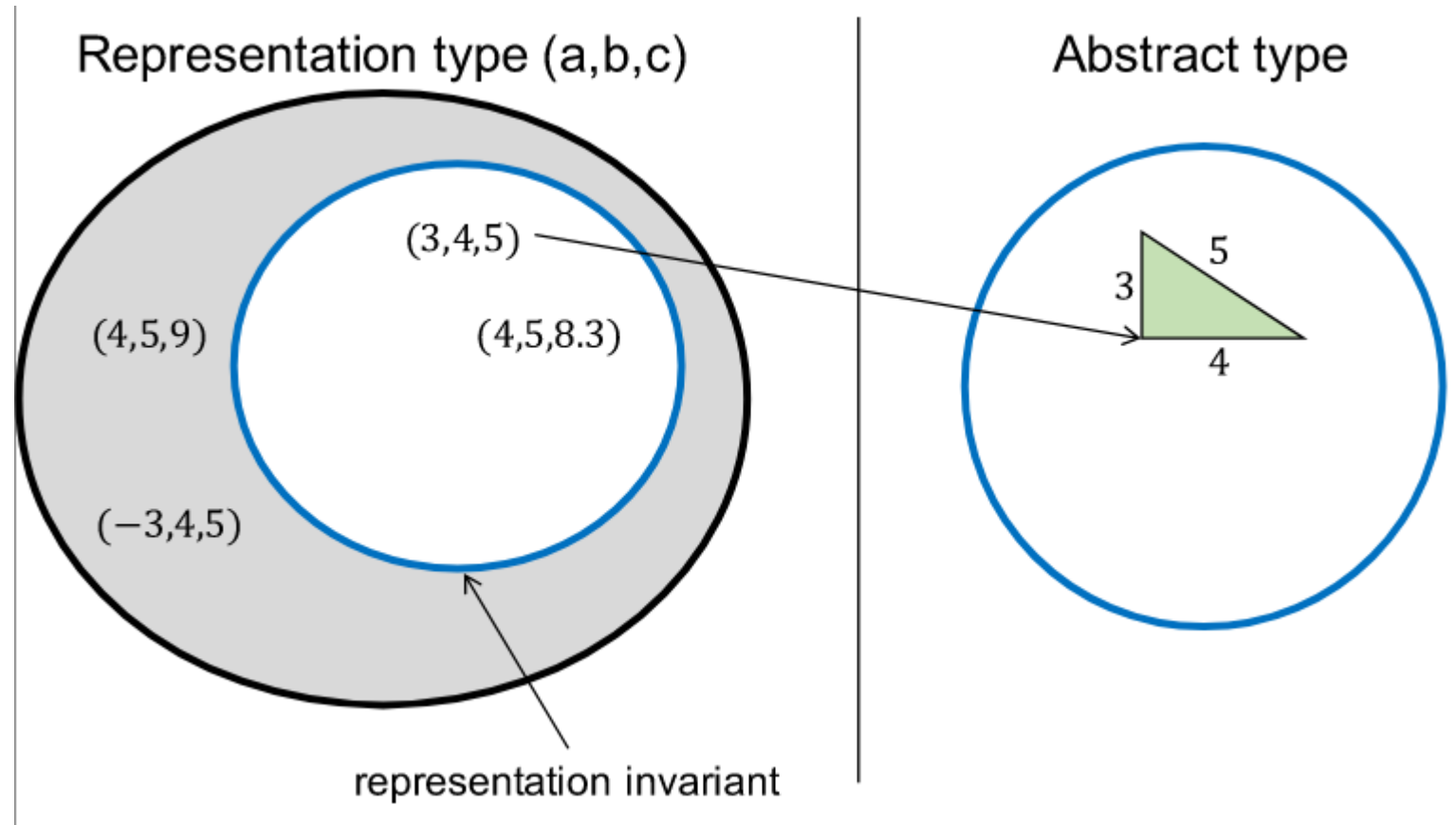
int main() {
    → Triangle t1; ↓
    Triangle_init(&t1, 3, 4, 5);
    Triangle_scale(&t1, 2);
}
```

# Invariantes & Asserções

# Invariantes da representação interna

- A **representação interna** usa **tipos pré-definidos** ou **outros** tipos definidos pelo programador
- Assegurar que os **valores dos atributos** representam instâncias / estados **válidos**
  - Por exemplo, o **comprimento do lado** de um triângulo não pode ser um **valor negativo**
- Definir **condições invariantes** !!

# Subconjunto das representações válidas



# Invariantes da representação interna

- Os invariantes estabelecem **condições** para que os **atributos** representem uma **instância válida**
- No caso da **1ª versão do TAD**
  - O **comprimento** de cada lado é um **valor positivo**
  - Os comprimentos dos lados satisfazem a **desigualdade triangular**
- Estabelecer e **documentar** os invariantes !!

# O TAD Triângulo – 1ª versão – Invariantes

```
// A triangle ADT.  
struct Triangle {  
    double a;  
    double b;  
    double c;  
    // INVARIANTS:  $a > 0 \ \&\& \ b > 0 \ \&\& \ c > 0 \ \&\&$   
    //  $a + b > c \ \&\& \ a + c > b \ \&\& \ b + c > a$   
};
```



# Asserções / Condições verdadeiras

- **Assegurar** os invariantes sempre que se **cria** ou **modifica** uma instância do TAD
- Usar **asserções de entrada / pré-condições**, sempre que possível, para **validar argumentos** das funções
- Assegurar os **invariantes à saída / pós-condições** de cada função que **modifique** uma **instância**
  - Pode não ser necessário fazê-lo explicitamente !!

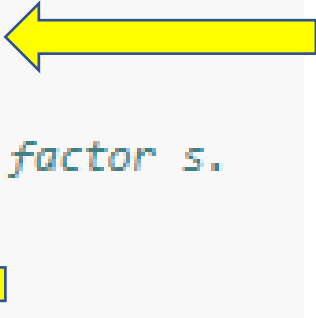
# Construtor – **Asserções de entrada**

```
// REQUIRES: tri points to a Triangle object;  
//           each side length is positive (a > 0 && b > 0 && c > 0);  
//           the sides meet the triangle inequality  
//           (a + b > c && a + c > b && b + c > a)  
// MODIFIES: *tri  
// EFFECTS:  Initializes the triangle with the given side lengths.  
void Triangle_init(Triangle *tri, double a, double b, double c) {  
    assert(a > 0 && b > 0 && c > 0);           // positive lengths  
    assert(a + b > c && a + c > b && b + c > a); // triangle inequality  
    tri->a = a;  
    tri->b = b;  
    tri->c = c;  
}
```



# Triangle\_scale – **Asserção de entrada**

```
// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS: Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
    assert(s > 0); // positive lengths
    tri->a *= s;
    tri->b *= s;
    tri->c *= s;
}
```



# Convenções & Testes

# Convenções habituais

- O utilizador de um TAD só opera com instâncias através da **interface do TAD**
  - I.e., as suas funções “**públicas**”
- O utilizador está, em geral, **proibido** de aceder diretamente aos campos da representação interna de cada instância
- **Esta convenção também é válida durante os testes do TAD**
  - Os testes avaliam o comportamento de um TAD e não a sua implementação

# Testar um TAD

- A **interface** define o modo como podemos interagir com as várias instâncias
  - Incluindo o código de teste do TAD !!
- **Modificar a representação / implementação** não deve implicar qualquer alteração do código de testes
  - **Executar**, de imediato, **testes** para verificar / validar as alterações efetuadas

# Testar um TAD

- Não é habitual / possível testar isoladamente cada função
  - Por exemplo, `Triangle_init()`
- Mas, podemos **testar** uma função construtora **em conjunto** com as funções de acesso aos atributos
- **Testar comportamento**, em vez de funções particulares !!

# O TAD Triângulo

– 2ª versão

## O TAD Triângulo – 2ª versão

- A **representação interna** de um **Triangle** é modificada
- **MAS** não é alterada a **interface** do TAD
- Não é necessário alterar a função **main()**
- O código que implementa as diversas funções **Triangle\_...** é alterado, **MAS** não o código que usa o TAD !!

# O TAD Triângulo – 2ª versão

```
// A triangle ADT.
struct Triangle {
    double side1;
    double side2;
    double angle;
};

// REQUIRES: tri points to a Triangle object
// MODIFIES: *tri
// EFFECTS: Initializes the triangle with the given side lengths.
void Triangle_init(Triangle *tri, double a_in,
                  double b_in, double c_in) {
    tri->side1 = a_in;
    tri->side2 = b_in;
    tri->angle = acos((pow(a_in, 2) + pow(b_in, 2) -
                        pow(c_in, 2)) /
                     (2 * a_in * b_in));
}
```

- Comprimento de dois lados
- Amplitude do ângulo formado

- Mesmos argumentos para o construtor



# “getters” – Alterar Triangle\_side3

```
// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the first side of the given Triangle.
double Triangle_side1(const Triangle *tri) {
    return tri->side1;
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the second side of the given Triangle.
double Triangle_side2(const Triangle *tri) {
    return tri->side2;
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the third side of the given Triangle.
double Triangle_side3(const Triangle *tri) {
    return tri->side3 sqrt(tri->side1 pow(tri->side1, 2) +
                        tri->side2 pow(tri->side2, 2) -
                        2 * tri->side1 * tri->side2 * tri->angle acos(tri->angle));
}
```

# Operações adicionais – Alterações

```
// REQUIRES: tri points to a valid Triangle  
// EFFECTS: Returns the perimeter of the given Triangle.  
double Triangle_perimeter(const Triangle *tri) {  
    return Triangle_side1(tri) + Triangle_side2(tri) + Triangle_side3(tri);  
}
```



```
// REQUIRES: tri points to a valid Triangle; s > 0  
// MODIFIES: *tri  
// EFFECTS: Scales the sides of the Triangle by the factor s.  
void Triangle_scale(Triangle *tri, double s) {  
    tri->side1 *= s;  
    tri->side2 *= s;  
}
```



# Organização em ficheiros

# Como organizar o código ?

- Como se podem alterar detalhes de representação / implementação sem alterar a interface ?
- Isolar a interface da implementação, usando 2 ficheiros separados
- Ficheiro **cabeçalho** + Ficheiro de **implementação**

# Organização em 2 ficheiros

- **Ficheiro cabeçalho .h** a ser **incluído** quando necessário
- Definição do tipo abstrato, de tipos de dados auxiliares e de constantes
- Definição dos **protótipos** das funções (públicas) da **interface**
- **Ficheiro de implementação .c**
- Concretização da **representação** do tipo abstrato
- Implementação das **funções da interface (públicas)**
- Implementação de **funções auxiliares (privadas)**

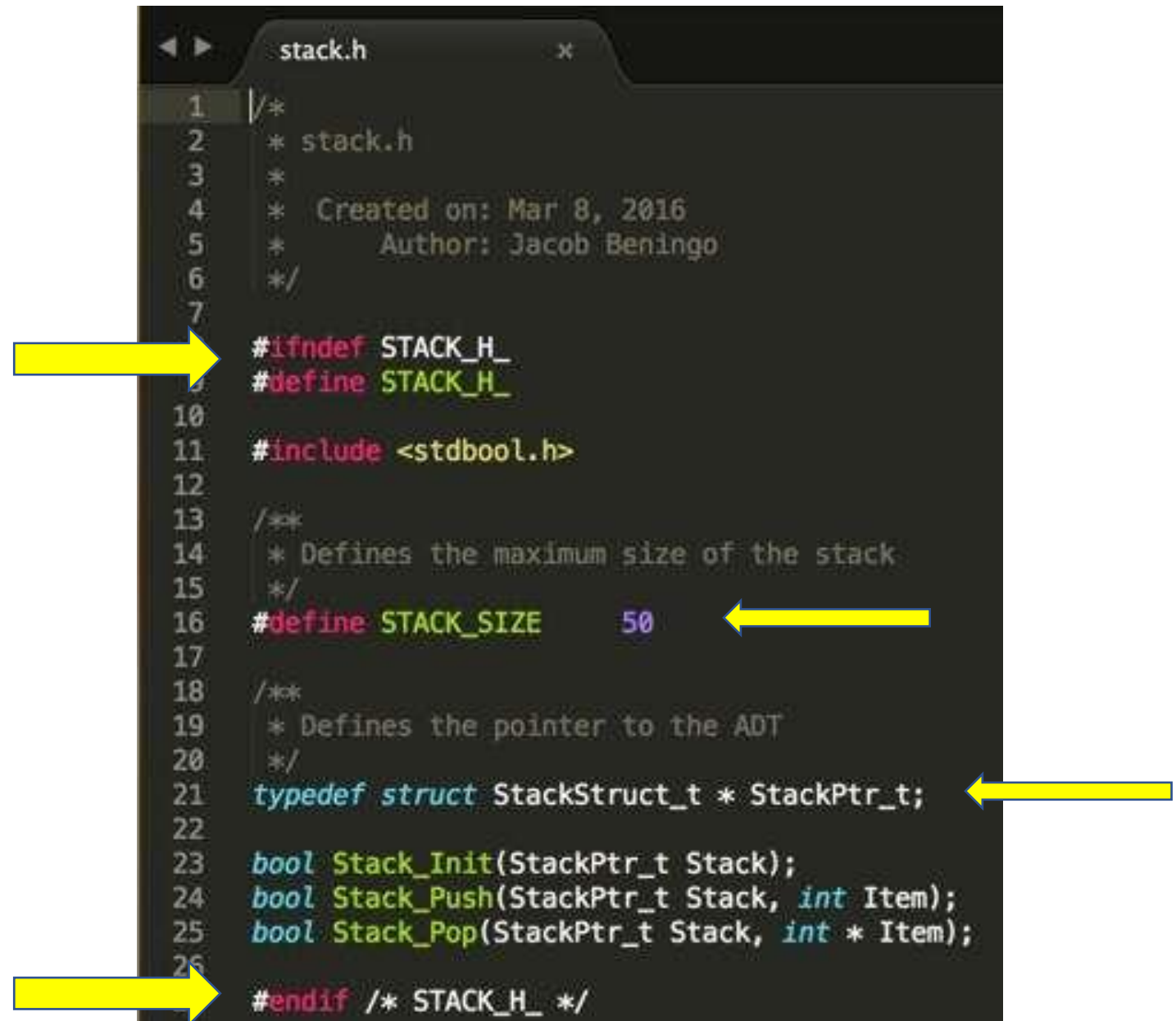
# Ficheiro .c

- A implementação pode variar durante o desenvolvimento de um projeto
- **OU** pode haver implementações diferentes para diferentes aplicações
- **Flexibilidade !!**

# Como evitar múltiplas inclusões / definições ?

- Se o mesmo **ficheiro .h** for **incluído em diferentes locais** de um programa, vai originar múltiplos **erros de definição**, em tempo de **compilação**
- Necesário usar as diretivas de processamento **#ifndef** (if not defined), **#define** e **#endif** no ficheiro .h
- Associar um **identificador único** a cada diretiva **#ifndef**
  - Convenção: usar **NOME-TAD\_H\_**

# Exemplo



```
1  /*
2  * stack.h
3  *
4  * Created on: Mar 8, 2016
5  * Author: Jacob Beningo
6  */
7
8  #ifndef STACK_H_
9  #define STACK_H_
10
11  #include <stdbool.h>
12
13  /**
14   * Defines the maximum size of the stack
15   */
16  #define STACK_SIZE 50
17
18  /**
19   * Defines the pointer to the ADT
20   */
21  typedef struct StackStruct_t * StackPtr_t;
22
23  bool Stack_Init(StackPtr_t Stack);
24  bool Stack_Push(StackPtr_t Stack, int Item);
25  bool Stack_Pop(StackPtr_t Stack, int * Item);
26
27  #endif /* STACK_H_ */
```

The image shows a code editor window titled 'stack.h'. The code is a C header file. Four yellow arrows point to specific lines: the first arrow points to line 8 (#ifndef STACK\_H\_), the second arrow points to line 16 (#define STACK\_SIZE 50), the third arrow points to line 21 (typedef struct StackStruct\_t \* StackPtr\_t;), and the fourth arrow points to line 27 (#endif /\* STACK\_H\_ \*/).



# Como evitar múltiplas inclusões / definições ?

- A **primeira vez** que uma dada diretiva **#ifndef** é encontrada
- O identificador associado **não** está definido
- O bloco de texto entre as diretivas **#ifndef** e **#endif** é processado
- E todos os **identificadores** desconhecidos aí encontrados ficam **definidos**

# Como evitar múltiplas inclusões / definições ?

- Da **próxima vez** que essa diretiva **#ifndef** for encontrada
- O identificador associado **já** está definido
- O bloco de texto entre as diretivas **#ifndef** e **#endif** é **ignorado**
- E não ocorrem quaisquer múltiplas definições **!!**

# Resumo

- TAD = especificação + interface + implementação
- Encapsular detalhes da representação / implementação
- Flexibilizar manutenção / reutilização / portabilidade
  
- Ficheiro .h : operações públicas + ponteiro para instância
- Ficheiro .c : implementação + representação interna



# Exemplo adicional

- O TAD Ponto 2D

# TAD Ponto 2D – 1ª versão – Coordenadas (x,y)

- **Tarefa:** desenvolver e testar o TAD Ponto2D
- Os **atributos** de cada ponto são as suas **coordenadas (x, y)**
- Que **funcionalidades** ?
- Que **testes** ?
- Que **representação** interna ?

# TAD Ponto 2D – 2ª versão – Coords. polares

- Uma representação interna alternativa é usar as **coordenadas polares (r, phi)** de cada ponto 2D.
- r é o **raio** (i.e., distância) relativamente à origem
- phi é o **ângulo** relativamente ao eixo horizontal
- Definição, desenvolvimento e teste faseados !!

# Especificação da interface



```
// A set of polar coordinates in 2D space.
struct Polar;

// REQUIRES: p points to a Polar object
// MODIFIES: *p
// EFFECTS: Initializes the coordinate to have the given radius and
//           angle in degrees.
void Polar_init(Polar* p, double radius, double angle);

// REQUIRES: p points to a valid Polar object
// EFFECTS: Returns the radius portion of the coordinate as a
//           nonnegative value.
double Polar_radius(const Polar* p);

// REQUIRES: p points to a valid Polar object
// EFFECTS: Returns the angle portion of the coordinate in degrees as
//           a value in [0, 360).
double Polar_angle(const Polar* p);
```

# 1º teste básico

```
// Basic test of initializing a Polar object.  
TEST(test_init_basic) {  
    Polar p;  
    Polar_init(&p, 5, 45);  
  
    ASSERT_EQUAL(Polar_radius(&p), 5);  
    ASSERT_EQUAL(Polar_angle(&p), 45);  
}
```





# Representação interna + Invariantes

```
struct Polar {  
    double r;  
    double phi;  
    // INVARIANTS: r >= 0 && phi >= 0 && phi < 360  
};
```



- Falta estabelecer uma condição / representação para a **origem**
- Por exemplo: **radius == 0 && phi == 0**

# 1ª implementação

```
void Polar_init(Polar* p, double radius, double angle) {  
    p->r = radius;  
    p->phi = angle;  
}  
  
double Polar_radius(const Polar* p) {  
    return p->r;  
}  
  
double Polar_angle(const Polar* p) {  
    return p->phi;  
}
```

# Casos de teste – Falham ?

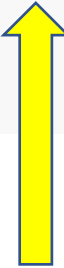
```
// Tests initialization with a negative radius.
TEST(test_negative_radius) {
    Polar p;
    Polar_init(&p, -5, 225);
    ASSERT_EQUAL(Polar_radius(&p), 5);
    ASSERT_EQUAL(Polar_angle(&p), 45);
}

// Tests initialization with an angle >= 360.
TEST(test_big_angle) {
    Polar p;
    Polar_init(&p, 5, 405);
    ASSERT_EQUAL(Polar_radius(&p), 5);
    ASSERT_EQUAL(Polar_angle(&p), 45);
}
```




# Melhorar a função construtora

```
void Polar_init(Polar* p, double radius, double angle) {  
    p->r = std::abs(radius); // set radius to its absolute value  
    p->phi = angle;  
    if (radius < 0) {          // rotate angle by 180 degrees if radius  
        p->phi = p->phi + 180; // was negative  
    }  
}
```

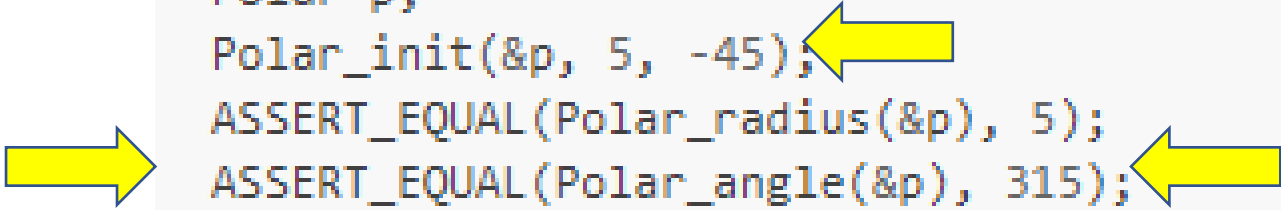


# Melhorar a função construtora



```
void Polar_init(Polar* p, double radius, double angle) {  
    p->r = std::abs(radius); // set radius to its absolute value  
    p->phi = angle;  
    if (radius < 0) {        // rotate angle by 180 degrees if radius  
        p->phi = p->phi + 180; // was negative  
    }  
    p->phi = std::fmod(p->phi, 360); // mod angle by 360  
}
```

# Testar de novo

```
// Tests initialization with a negative angle.  
TEST(test_negative_angle) {  
    Polar p;  
    Polar_init(&p, 5, -45);  
    ASSERT_EQUAL(Polar_radius(&p), 5);  
    ASSERT_EQUAL(Polar_angle(&p), 315);  
}
```



# Melhorar a função construtora

```
void Polar_init(Polar* p, double radius, double angle) {  
    p->r = std::abs(radius); // set radius to its absolute value  
    p->phi = angle;  
    if (radius < 0) {          // rotate angle by 180 degrees if radius  
        p->phi = p->phi + 180; // was negative  
    }  
    p->phi =  fmod(p->phi, 360); // mod angle by 360  
 if (p->phi < 0) {          // rotate negative angle by 360  
        p->phi += 360;  
    }  
}
```



# Exercícios / Tarefas



# Tarefa – 2 versões do TAD Ponto 2D

- Especificar a **interface** para o TAD Ponto 2D
- Desenvolver as **duas implementações** distintas
  - Coordenadas cartesianas
  - Coordenadas polares
- A interface é a mesma para ambas as implementações
- Desenvolver **um único programa de teste**