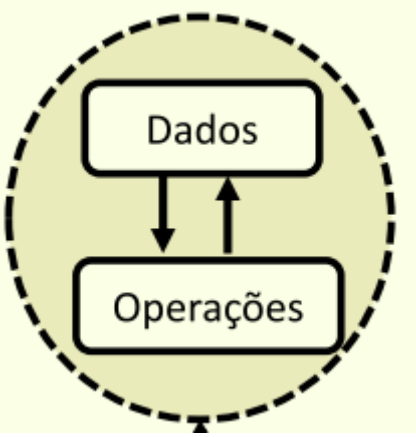
Passos para o desenvolvimento de um TAD

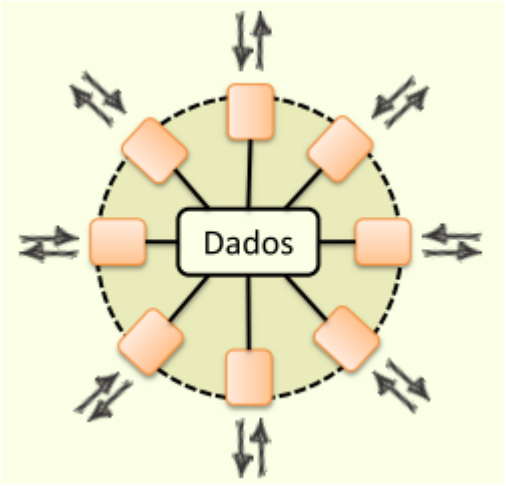
Introdução

Em diversas situações, os tipos de dados fornecidos pela linguagem podem não atender satisfatoriamente algumas necessidades da nossa aplicação. Nesse sentido, podemos definir um Tipo Abstrato de Dados (TAD) para satisfazer tal necessidade. Um TAD é uma forma de definir um novo tipo de dado juntamente com as operações que manipulam esse novo tipo.

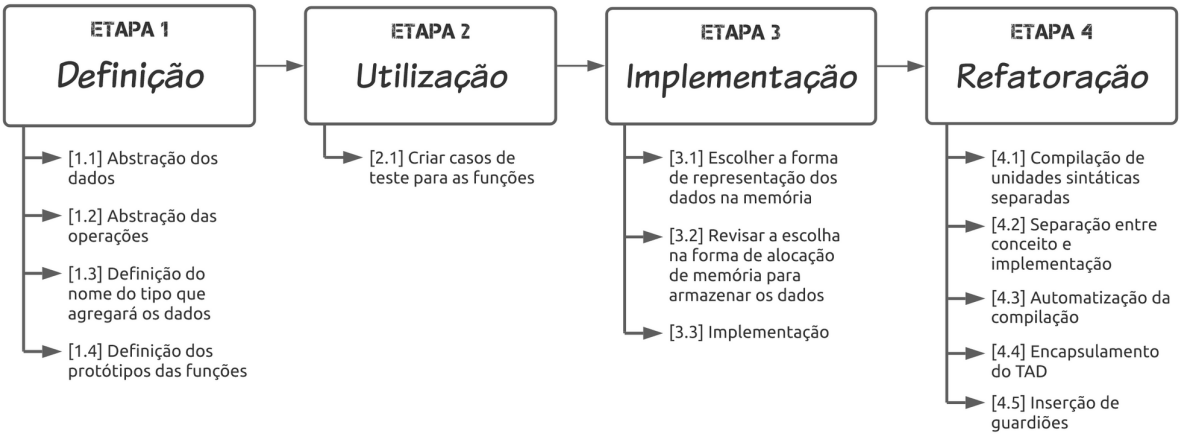


Para desenvolver um TAD, devemos ter em mente algumas características importantes sobre ele.

- Reutilização. quanto mais independente (acoplamento fraco) o TAD for, mais reutilizável ele será. Essa característica é importante para que possamos reutilizar esse tipo sempre que precisarmos, tal como uma biblioteca.
- Separação entre conceito e implementação. Em outras palavras, separação da definição do tipo e a sua implementação. As aplicações que utilizarão o TAD somente precisam saber quais informações o tipo armazena e quais operações estão disponíveis. Os detalhes de implementação não interessam. Esse conceito está presente em diferentes contextos, por exemplo, no preparo de uma vitamina de frutas, quando você for operar um liquidificador, você só precisa saber o que faz cada botão, os detalhes mecânicos e eletrônicos são irrelevantes. Quando vamos construir um TAD precisamos pensar em quem vai utilizá-lo.



Para guiar o nosso aprendizado, vamos utilizar um estudo de caso descrevendo o desenvolvimento de um TAD em 4 etapas: **definição**, **utilização**, **implementação** e **refatoração**.



Estudo de Caso

Como estudo de caso, vamos imaginar que estamos desenvolvendo uma aplicação e nos deparamos com a necessidade de representar e manipular datas. Como não existe nenhum tipo específico para representar uma data na linguagem C, vamos criar um tipo abstrato de dados que atenda nossa necessidade.

Etapa 1: Definição

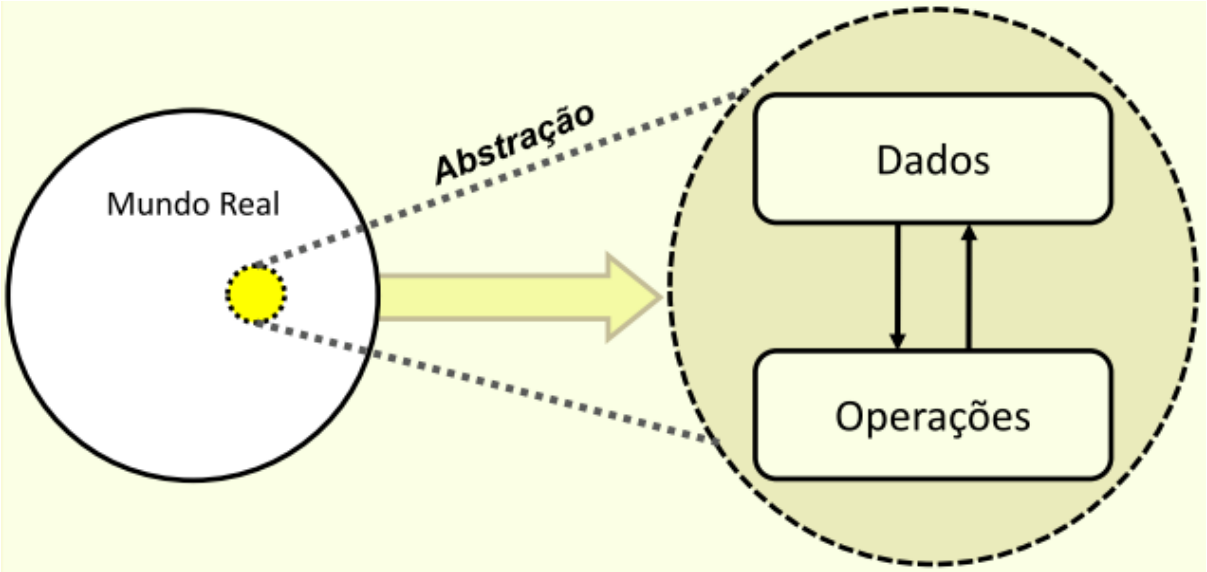
Antes de iniciar a definição do TAD, vamos primeiro ter clareza a respeito do conceito de abstração. Resumidamente, a abstração é a habilidade de concentrar-se apenas nos aspectos essenciais do contexto observado, ignorando as características menos relevantes. Na computação, essa habilidade facilita a vida do programador, pois simplifica a solução de problemas complexos.

O TAD tem como objetivo representar computacionalmente um objeto do mundo real por meio da abstração, ou seja, concentrando-se nas informações mais importantes e ignorando detalhes irrelevantes. A pergunta que devemos ter em mente ao definir um TAD é: **O QUE eu preciso?** Nesta etapa não devemos nos preocupar em **COMO** vamos implementar, e sim **O QUE** precisamos.

O TAD é composto por duas partes bem definidas: **dados e operações**. Os dados consistem nas informações que precisamos representar computacionalmente e as operações retratam quais funcionalidades sobre os dados serão disponibilizadas. O processo de abstração deve ser aplicado para cada uma dessas partes.

a) **Abstração dos dados**: é a forma de ver um dado/objeto a ser modelado sem se preocupar em como ele será organizado na memória. Por exemplo, se o objeto em questão é um Aluno, eu só preciso saber quais são informações do aluno serão representadas (Registro Acadêmico, Nome, CPF, etc). Detalhes de como essas informações serão organizadas na memória são irrelevantes neste momento.

b) **Abstração funcional**: é a forma de ver uma operação sem se preocupar com os detalhes de implementação. **Eu preciso saber** O QUE a operação deve fazer, quais são as informações de ENTRADA e qual a SAÍDA produzida. Neste momento, não nos interessa saber os detalhes de **COMO** a operação transformará os dados da entrada na saída esperada.



Nós já utilizamos implicitamente esse conceito nos tipos primitivos da linguagem de programação. Por exemplo, quando vamos utilizar um dado do tipo inteiro, somente precisamos saber quais operações podem ser realizadas com esse tipo de dado (operações aritméticas). Não precisamos dos detalhes de como o número inteiro é armazenado na memória e de como a operação de multiplicação irá manipular o dado. Consegue enxergar essa similaridade?

Portanto, na etapa de definição do TAD, vamos aplicar as abstrações citadas anteriormente seguindo dois passos:

- [ ] **Passo 1.1 - Abstração dos dados**. Precisamos listar todas as informações que precisam ser modeladas computacionalmente. Seguindo o nosso estudo de caso, precisamos "olhar" para uma data e decidir quais informações serão modeladas. Para o nosso exemplo, vamos precisar do dia, mês e ano. Veja que o horário também poderia ser incluído nessa modelagem.
- [ ] **Passo 1.2 - Abstração das operações**. Precisamos listar todas as operações que serão disponibilizadas para a data. No nosso exemplo, vamos listar apenas algumas operações.

Resultado dos passo 1.1 e 1.2 do nosso estudo de caso:

```
Resultado do Passo 1.1: Abstração dos dados
- Dia
- Mês
- Ano

Resultado do Passo 1.2: Abstração das operações:
Operação 1: Criar uma data
Operação 2: Destruir uma data
Operação 3: Duplicar uma data
Operação 4: Alterar uma data
Operação 5: Recuperar o dia da data
Operação 6: Recuperar o mês (número) da data
Operação 7: Recuperar o ano da data
Operação 8: Recuperar a data na forma de uma cadeia de caracteres seguindo o formato "dia/mês/ano"
Operação 9: Recuperar o mês da data por extenso
```

Definido **O QUE** é preciso, agora precisamos introduzir essas necessidades no nosso TAD. Cada funcionalidade será traduzida para uma função no TAD. Mas antes das funções, precisaremos definir um nome para a estrutura que agregará todos os dados do TAD.

- [ ] **Passo 1.3 - Definição do nome do tipo que agregará os dados**. Vamos criar registro vazio (para posteriormente agregar as informações) e definir um novo nome para esse tipo por meio da diretiva typedef. Criaremos um registro vazio porque não queremos, por enquanto, nos preocupar em como as informações serão organizadas na memória. Até agora, sabemos que o novo tipo se chama Data e que ele armazenará as informações do dia, mês e ano.

```
typedef struct{

}Data;
```

- [ ] **Passo 1.4 - Definição dos protótipos das funções**. Nessa etapa precisamos criar um protótipo de função para cada operação listada. Temos em mãos a descrição das operações e o nome do tipo a ser manipulado (Data). Diante disso, precisamos definir:
    - **O nome da função**

    - **Saída esperada**. Qual é o resultado a ser produzido pela função? Veja que não precisamos especificar como fazer. Por exemplo, o resultado esperado da operação 8 é uma cadeia de caracteres contendo o nome do mês armazenado na data.
    - **Parâmetros de entrada**. Quais informações são necessárias para que a função produza o resultado esperado?

Faremos isso para cada operação descrita. Embora ainda não precisamos nos preocupar com a implementação, nessa etapa precisamos definir a forma de alocação do TAD (estática ou dinâmica). Essa definição é necessária para especificar a saída esperada e os parâmetro de entrada.

Por exemplo, considere a operação 8 que recupera a data na forma de uma cadeia de caracteres.

- Como a função devolverá essa cadeia de caracteres?
    - A função será responsável por alocar dinamicamente os espaço para acomodar a cadeia de caracteres? ou
    - A função não assumirá a responsabilidade de alocação e solicitará um endereço com o espaço já previamente alocado? Essa opção necessita que o endereço seja solicitado nos parâmetros de entrada, pois é um requisito para que a função exerça sua responsabilidade.

Tal como esse exemplo, outras funções também dependem da forma de alocação de memória. Portanto, precisaremos fazer uma escolha nesse momento, mas nada impede que você altere isso durante a implementação.

Agora, vamos criar um protótipo de função para cada operação especificada

```c
//Operação 1: Criar uma data
Data* data_criar(int dia, int mes, int ano);
// outra possibilidade
// void data_criar(Data* d, int dia, int mes, int ano);

//Operação 2: Destruir uma data
void data_desalocar(Data** d);
// outra possibilidades
// void data_destruir(Data* d);

//Operação 3: Duplicar uma data
Data* data_clone(Data* d);
// outra possibilidade
// void data_clone(Data* destino, Data* origem);

//Operação 4: Alterar uma data
void data_alterar(Data* d, int dia, int mes, int ano);

//Operação 5: Recuperar o dia da data
int data_dia(Data* d);

//Operação 6: Recuperar o mês (número) da data
int data_mes(Data* d);

//Operação 7: Recuperar o ano da data
int data_ano(Data* d);

//Operação 8: Recuperar a data na forma de uma cadeia de caracteres seguindo o formato "dia/mês/ano"
int data_string(Data* d, char* string);
// outra possibilidade
// char* data_string(Data* d);


//Operação 9: Recuperar o mês da data por extenso
int data_mesExtenso(Data* d, char* str);
// outra possibilidade
// char* data_mesExtenso(Data* d);
```

Resultado final da Etapa 1.

```c
/*****************************************************************
*   DADOS
*****************************************************************/
typedef struct{

}Data;

/*****************************************************************
*   OPERACOES
*****************************************************************/
Data* data_criar(int dia, int mes, int ano);
void data_desalocar(Data** d);
Data* data_clone(Data* d);
void data_alterar(Data* d, int dia, int mes, int ano);
int data_dia(Data* d);
int data_mes(Data* d);
int data_ano(Data* d);
char* data_string(Data* d);
char* data_mesExtenso(Data* d);
```

Somente essas informações são suficientes para que o programador utilize o TAD. Para demostrar isso na prática, a seguir, vamos fazer a utilização desse TAD sem conhecer sua implementação.

Etapa 2: Utilização

A utilização das funções sem conhecer sua implementação também é uma ótima forma de o auxiliar na implementação. Pense comigo: se você for capaz de usar a função, significa que você compreendeu o que ela faz e isso te dará clareza na implementação. Entretanto, se você não for capaz de usar a função, significa que não está claro na sua cabeça o que a função faz e, muito provavelmente, você terá dificuldades na implementação.

> Se você não sabe para onde ir, qualquer caminho serve.

Para enriquecer ainda mais essa etapa, podemos utilizar as funções na criação de casos de teste para elas. Isso ampliará significativamente sua compreensão sobre a responsabilidade de cada função e te auxiliará na captura de eventuais erros durante a implementação.

- [ ] **Passo 2.1 - Criar casos de teste para as funções.** Nesse passo, tente criar um caso de teste para um agrupamento pequeno de funções. Se possível, crie um caso de teste para cada função.

Vamos utilizar a biblioteca assert.h para nos auxiliar na criação dos casos de testes. Essa biblioteca traz a definição da função assert() que testa se uma expressão é verdadeira. Se a expressão passada por parâmetro para a função assert()for verdadeira, o assertse mantém silencioso (ele não acusará nada). Caso a expressão for falsa, o assert acusará erro e abortará a execução. Por exemplo, a chamada assert(1==1) executará sem problemas. Já a chamada assert(1>2) interromperá a execução pois a expressão 1>2 é falso.

Vamos elaborar um teste para as funções data_criar e data_string utilizando o assert.

```c
#include <assert.h>

int main(){
    // Criamos a data 01/10/2020
    Data* data1 = data_criar(01,10,2020);
    char dataStr[15];
    // Copiamos a versão string da data para a variável dataStr.
    data_string(data1, dataStr);
    // Neste ponto, o resultado esperado é que o conteúdo da variável dataStr seja "01/10/2020".
    // Dessa forma, vamos utilizar o assert para verificar esse comportamento.
    assert(strcmp(dataStr, "01/10/2020") == 0);

}
```

Veja o resultado final de um exemplo de teste das funções do TAD.

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>

/*****************************************************************
*   DADOS
*****************************************************************/
typedef struct data{

}Data;

/*****************************************************************
*   OPERACOES
*****************************************************************/
Data* data_criar(int dia, int mes, int ano);
void data_desalocar(Data** enderecoData);
Data* data_clone(Data* d);
int data_alterar(Data* d, int dia, int mes, int ano);
int data_dia(Data* d);
int data_mes(Data* d);
int data_ano(Data* d);
int data_string(Data* d, char* str);
int data_mesExtenso(Data* d, char* str);


/*****************************************************************
*   UTILIZACAO - Comportamento esperado
*****************************************************************/
int main(){
    printf("\nTESTE INICIALIZADO\n");

    /*****************************************************************
    * FUNÇÕES TESTADAS :
    *  - data_criar: Criar data
    *  - data_string: Recuperar a data na forma de uma cadeia de caracteres seguindo
    *    o formato "dia/mês/ano"
    *****************************************************************/
    Data* data1 = data_criar(01,10,2020);
    char dataStr[15];
    data_string(data1, dataStr);
```

```
    // O conteúdo da variável dataStr é comparada com a saída esperada "01/10/2020"
    printf("- Testando data_criar \n");
    printf("- Testando data_string \n");
    assert(strcmp(dataStr, "01/10/2020") == 0);

    /*****************************************************************
    * FUNÇÕES TESTADAS :
    *  - data_dia: Recuperar o dia da data
    *  - data_mes: Recuperar o mês da data
    *  - data_ano: Recuperar o ano da data
    *****************************************************************/

    printf("- Testando data_dia \n");
    int dia = data_dia(data1);
    assert(dia == 1);

    printf("- Testando data_mes \n");
    int mes = data_mes(data1);
    assert(mes == 10);

    printf("- Testando data_ano \n");
    int ano = data_ano(data1);
    assert(ano == 2021);

    /*****************************************************************
    * FUNÇÕES TESTADAS :
    *  - data_clone: Duplicar uma data
    *  - data_alterar: Alterar uma data
    *****************************************************************/
    Data* copia = data_clone(data1);
    data_alterar(copia, 31, 12, 2021);

    //alteramos a copia e verificamos se a alteração não impactou a data original
    printf("- Testando data_clone \n");
    printf("- Testando data_alterar \n");
    assert(data_dia(data1) == 1);
    assert(data_dia(copia) == 31);
    // verificamos se o mes e o ano também foram alterados
    assert(data_mes(copia) == 12);
    assert(data_ano(copia) == 2021);
```

```
    /*****************************************************************
    * FUNÇÕES TESTADAS :
    *  - data_mesExtenso: Recuperar o mês da data por extenso
    *****************************************************************/
    printf("- Testando data_mesExtenso \n");
    char str[15];
    data_mesExtenso(data1, str);
    assert(strcmp(str, "outubro") == 0);

    /*****************************************************************
    * FUNÇÕES TESTADAS :
    *  - data_desalocar: Destruir uma data
    *****************************************************************/
    printf("- Testando data_desalocar \n");
    data_desalocar(&data1);
    data_desalocar(&copia);
    assert(data1 == NULL);
    assert(copia == NULL);

    printf("TESTE FINALIZADO\n");
    return 0;
}
```

Etapa 3: Implementação

A implementação de um tipo abstrato de dados é o momento para responder todas as perguntas da perspectiva COMO.

- Como os dados devem ser modelados e organizados na memória?
- Como os dados devem ser manipulados por suas operações?
- Como as operações transformarão os parâmetros de entrada nos resultados esperados?

Vamos fazer um *checklist* com as etapas básicas da implementação de um TAD:

- [ ] **Passo 3.1 - Escolher a forma de representação dos dados na memória**. Essa etapa é importante, pois todo o resto depende dessa decisão. Para auxiliar nessa decisão, precisamos estar bem familiarizados com os tipos primitivos e os mecanismos de agregação de dados (tipos compostos) existentes na linguagem de programação utilizada. Portanto, certifique-se que você domina os conceitos fundamentais, tais como vetores, registros e ponteiros. Caso necessário, revise o conteúdo das aulas anteriores para solidificar esses conceitos fundamentais e facilitar o aprendizado desse novo conteúdo.

Para o nosso exemplo, vamos armazenar a data na forma de 3 números inteiros. Portanto, o registro Data terá três atributos. Veja que essa não é a única forma de representar uma data na memória. Poderíamos ter optado por um vetor de inteiros, ou um vetor de char. A escolha é sua. Para quem utilizará sua TAD, essa escolha e irrelevante, pois ele está interessado nas operações.

A implementação das funções dependem dessa escolha. Perceba que nas funções, nós sempre passamos o endereço do registro Data. E, por meio desse endereço, as funções poderão acessar os dados da data. Por exemplo, com a referência Data* d, a função pode acessar o dia com a instrução d->dia. Caso a escolha de organização da data fosse um vetor, a instrução para acessar o dia seria d->vetor[0].

Uma característica importante proporcionada pela abstração é a facilidade de manutenção. Se mantivermos os protótipos das funções (também chamado de contrato ou interface) inalterados, podemos realizar alterações na organização dos dados e no código das funções sem impactar os programas que usam o nosso TAD. Um exemplo fora de contexto que ressalta essa característica é quando uma pessoa troca o carro movido a combustível por um elétrico. Embora internamente haja diferenças significativas no funcionamento, uma vez que a interface do carro é mantida (volante, acelerador, freio, embreagem, câmbio, etc), não é necessário nenhuma modificação na forma de dirigir.

```c
/**************************************************************************
*   DADOS
**************************************************************************/
typedef struct data{
    int dia;
    int mes;
    int ano;
}Data;
```

- [ ] **Passo 3.2 - Revisar a escolha na forma de alocação de memória para armazenar os dados**. Como já discutido no passo 1.4, ao definir o protótipo da função, é preciso definir a forma de alocação de memória. Portanto, nesse passo, apenas vamos revisar se a função terá como responsabilidade alocar espaço ou se solicitará espaço previamente alocado.
- [ ] **Passo 3.3 - Implementação.** Com a representação dos dados definida e os protótipos revisados, agora é hora de por a mão na massa, ou melhor, os dedos no teclado. Vamos implementar as operações definidas nos protótipos levando em consideração a eficiência dos algoritmos.

```c
/**************************************************************************
*   IMPLEMENTACAO
**************************************************************************/
Data* data_criar(int dia, int mes, int ano){
    Data* d = (Data*) calloc(1,sizeof(Data));
    d->dia = dia;
    d->mes = mes;
    d->ano = ano;
    return d;
}

void data_desalocar(Data** enderecoData){
    free(*enderecoData);
    *enderecoData = NULL;
}

Data* data_clone(Data* d){
    if(d==NULL) return NULL;

    Data* novo = data_criar(d->dia, d->mes, d->ano);
    return novo;
}

int data_alterar(Data* d, int dia, int mes, int ano){
    if(d == NULL) return 0;

    d->dia = dia;
    d->mes = mes;
    d->ano = ano;
    return 1;
}

int data_dia(Data* d){
    if(d == NULL) return -1;

    return d->dia;
}
```

```c
int data_mes(Data* d){
    if(d == NULL) return -1;
    return d->mes;
}
int data_ano(Data* d){
    if(d == NULL) return -1;
    return d->ano;
}

int data_string(Data* d, char* str){
    if(d == NULL) return 0;

    sprintf(str, "%02d/%02d/%d", d->dia, d->mes, d->ano);
    return 1;
}
int data_mesExtenso(Data* d, char* str){
    if(d == NULL) return 0;

    char meses[12][10] = {"janeiro","fevereiro","marco","abril","maio","junho",
    "julho","agosto","setembro","outubro","novembro","dezembro"};
    strcpy(str, meses[d->mes-1]);
    return 1;
}
```

Dica: Adquira o hábito de refatorar suas funções. Primeiro faça funcionar, mas não esqueça de revisitar o código em busca de oportunidades de melhorar a legibilidade e eficiência dos seus algoritmos.

Etapa 4: Refatoração

> Para fins didáticos, vamos dividir a refatoração em diferentes etapas. No entanto, com a prática, você realizará a refatoração sem a necessidade dessa divisão.

4.1 - Compilação de unidades sintáticas separadas

Vamos separar a implementação do TAD da sua utilização na função main. Essa separação permitirá a reutilização do TAD em outros programas.

- [ ] Criar 2 arquivos
    - tad_data.h para o TAD
    - main.c para a utilização do TAD
- [ ] Copiar a função main para o arquivo main.c
- [ ] Copiar a definição da *struct*, os protótipos e a implementação para o tad_data.h

4.2 - Separando conceito da implementação

Vamos dividir o código do TAD em 2 arquivos para separar a definição da implementação. Essa separação proporcionará a abstração mencionada inicialmente.

O arquivo tad_data.h conterá a definição do TAD e o arquivo tad_data.c conterá a implementação.

- [ ] Criar o arquivo tad_data.c e mover a implementação das funções
- [ ] Incluir a definição no cabeçalho do arquivo tad_data.c: #include "tad_data.h"

Veja que, a partir dessa modificação, a compilação não funciona mais.

```
gcc main.c -o main
```

Precisamos compilar a implementação tad_data.c separadamente (gerando o arquivo objeto tad_data.o) e incluir esse arquivo objeto na compilação do arquivo main.c

```
gcc -c tad_data.c -o tad_data.o
gcc main.c tad_data.o -o main
```

4.3 - Automatizando a compilação

Vamos automatizar a compilação usando o makefile.

- [ ] Crie um arquivo makefile com a seguinte estrutura:

```
all:

run:

clean:
```

- [ ] Na macro all, vamos colocar as instruções de compilação

```
gcc -c tad_data.c -o tad_data.o
gcc main.c tad_data.o -o main
```

- [ ] Na macro run, vamos colocar a instrução de execução

```
./main
```

- [ ] Na macro clean, vamos colocar instruções que apagarão os arquivos gerados na compilação

```
rm *.o
rm main
```

O arquivo final ficará assim

```
all:
  @echo "compilando"
  gcc -c tad_data.c -o tad_data.o
  gcc main.c tad_data.o -o main

run:
  ./main

clean:
  rm *.o
  rm main
```

Utilize tabulações ao invés de espaço. Se você usar espaços o make acusará problemas.

Para compilar use: make

Para executar use: make run

Para apagar os arquivos gerados use: make clean

Podemos também parametrizar o makefile

```
MAIN=main
TAD=tad_data

all:
  @echo "compilando"
  gcc -c $(TAD).c -o $(TAD).o
  gcc $(MAIN).c $(TAD).o -o $(MAIN)

run:
  ./$(MAIN)

clean:
  rm *.o
  rm $(MAIN)
```

4.4 - Encapsulando o TAD

Vamos aplicar o conceito de encapsulamento na implementação do TAD para

- "esconder" o conteúdo da *struct* e eventuais funções auxiliares.
- [ ] Copiar a *struct* Data para a implementação (tad_data.c) com o seguinte formato:

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

- [ ] Substituir a declaração da *struct* na definição (tad_data.h) por:

```
typedef struct data Data;
```

- [ ] Para verificar o encapsulamento na prática, vamos criar uma função de validação da data que somente será usado internamente e não será disponibilizado na sua interface.

```
int dataValida(int dia, int mes, int ano){
    if(dia < 1 || dia > 31) return 0;
    if(mes < 1 || mes > 12) return 0;

    return 1;
}
```

- [ ] Aplique essa validação nas funções data_criar e data_alterar
- [ ] Verifique as características do encapsulamento. A IDE não exibe mais os atributos da *struct* Data bem como a função dataValida.

4.5 - Inserindo guardiões para evitar sobreposição de definições

Vamos utilizar guardiões disponibilizado pela linguagem para evitar sobreposição nas inclusões.

- [ ] Envolver o código da definição tad_data.h com o guardião

```
#ifndef _TAD_DATA_H_
#define _TAD_DATA_H_
  ...
#endif
```

Esse guardião evitará que o TAD seja incluído várias vezes.

Síntese

Etapa 1: Definição do TAD

- 1.1 - Abstração dos dados
- 1.2 - Abstração das operações.
- 1.3 - Definição do nome do tipo que agregará os dados
- 1.4 - Definição dos protótipos das funções

Etapa 2: Utilização do TAD

- 2.1 - Criar casos de teste para as funções

Etapa 3: Implementação do TAD

- 3.1 - Escolher a forma de representação dos dados na memória
- 3.2 - Revisar a escolha na forma de alocação de memória para armazenar os dados
- 3.3 - Implementação

Etapa 4: Refatoração

- 4.1 - Compilação de unidades sintáticas separadas
- 4.2 - Separação entre conceito e implementação
- 4.3 - Automatização da compilação
- 4.4 - Encapsulamento do TAD
- 4.5 - Inserção de guardiões para evitar sobreposição de definições

Referências

- Estrutura de Dados e Técnicas de Programação. Francisco Bianchi
- Capítulo 12 do livro Introdução a Estruturas de Dados - Com Técnicas de Programação em C. (2a edição).
- Capítulo 2 do livro Estrutura de Dados da UFRGS