



Estrutura de Dados

```
3 Estrutura de dados cliente:
4     char rua[40];
5     int numero;
6     char complemento[30];
7     char bairro[20];
8     char cidade[20];
9     char estado[2];
10    int cep;
11 };
12
13 typedef struct dadosCliente cliente; //Posso utilizar o mesmo nome da struc
14 Estrutura dadosCliente cliente; //Posso utilizar o mesmo nome da struc
15     char nome[30];
16     int idade;
17     float salario;
18     char genero;
19     dadosEnderecoend; //Tipo previamente declarado neste código
20 };
21
22 int main() {
23     printf("\nInício do código.");
24     cliente cli[2]; //O vetor cli é do tipo cliente
25     int i;
26
27     //Preencher alguns itens do cliente, para exemplificar.
28     for (i = 1; i <= 2; i++) {
29         printf("\nDigite o salário do cliente: ");
30         scanf("%f", &cli[i].salario); //Recebe a digitação do sal
31
32         printf("\nDigite a cidade do cliente");
33         get(cli[i].end.cidade);
34
35     }
36 }
```



TIPOS ABSTRACTOS DE DADOS

APRESENTAÇÃO

O lá aluno (a), sempre que precisamos armazenar dados em nossos códigos devemos avaliar se ele é de algum tipo primitivo da linguagem, ou seja, devemos verificar se há algum tipo de dado que atenda a este armazenamento.

Mas você já se perguntou se existe alguma possibilidade de unir um ou mais tipos para utilização? Assim, ao invés de armazenarmos um item por vez, faríamos o armazenamento de várias informações.

Pois já te digo que temos como fazer isso, vamos ver como?

OBJETIVOS DE APRENDIZAGEM

Ao final desse módulo você deverá ser capaz de:

- Relacionar tipos primitivos com tipos abstratos de dados (TAD);
- Construir códigos que possuam a aplicação de TAD;
- Construir TAD's a partir de abstrações do mundo real.

TIPOS ABSTRATOS DE DADOS

Introdução

Imagine que você irá armazenar os dados referente ao endereço de um cliente, e neste caso, você terá que incluir nome da rua, número, complemento, bairro, cidade, etc. Acredito que neste momento você deve ter considerado cada variável que precisará armazenar correto? Pois é, isso seria bem trabalhoso, várias variáveis e informações "soltas".

Para te poupar de tanto trabalho, vamos falar sobre os Tipos Abstratos de Dados (TAD's), eles são responsáveis pelo agrupamento de vários tipos em um mesmo item, criando assim uma única entidade para que seja utilizada com suas especificações para cada necessidade que você tiver... bem vamos entender isso com mais calma.



Para te explicar melhor vamos voltar ao nosso exemplo original, onde criaremos variáveis para receber as informações de endereço de um cliente, neste caso, as exigências serão:

Rua
Número
Complemento
Bairro
Cidade
Estado
CEP

Agora imaginaremos o que será necessário para armazenar cada item, no programa:

Rua - String com 40 posições
Número - Inteiro
Complemento - String com 30 posições
Bairro - String com 20 posições
Cidade - String com 20 posições
Estado - String com 2 posições
CEP - Inteiro

OBSERVAÇÃO

Observe que, em sua memória, cada variável teria um armazenamento diferente, e, além disto, se precisássemos de informações de vários clientes, precisaríamos de um vetor de cada uma das informações, o que geraria uma verdadeira "loucura" para manter, afinal, teríamos que acessar cada item separadamente para utilizar.



MEMÓRIA

	Rua		Complemento		Bairro
0	AX00	0	LI19	0	KG38
1	AX01	1	LI20	1	KG39
2	AX02	2	LI21	2	KG40
...
39	AX39	29	AX48	19	KG57

	Cidade		Estado		Numero
0	EW00	0	RN13		GF88
1	EW01	1	RN14		
2	EW02				
...	...				
19	EW19				WP05

É aí que aplicamos o conceito de Tipo Abstrato de Dados (TAD), a partir dele o programador poderá unir vários dados em um só “tipo” e, ao invés de acessar cada item, acessaria apenas o tipo pré definido anteriormente e o membro referente à informação interessada. Então, em sua memória, ficaria assim:

MEMÓRIA

	Rua		Complemento		Bairro
0		0		0	
1		1		1	
2		2		2	
...		
39		29		19	

	Cidade		Estado		Numero
0		0			HI00
1		1			...
2					
...					
19					CEP

Até o número de bytes necessários para armazenar tudo.



Observe que agora você não precisa mais manter as variáveis separadas, pois elas foram encapsuladas em uma única entidade, portanto, seu acesso será ao item principal, a partir daí, você informará qual membro deseja utilizar.

Esse é o conceito principal que você precisa saber, agora vamos aprender como criar cada item deste, para então colocarmos em prática!



typedef

Para facilitar a declaração, e passarmos a ter um “tipo” que foi definido em nosso código, utilizaremos o comando *typedef*, ele será responsável por criar tipos a partir de características definidas pelo usuário.

REFLITA

Imagine que você precisa ter um tipo específico, com características e nome definidos por você, então, para isso, você precisará utilizar o *typedef*.



sintaxe do comando é:

```
typedef características nome_novo;
```

Em características você definirá tudo o que será relevante para este novo tipo, criado por você e em nome_novo indique qual será o apelido (*alias*) utilizado para nomear o tipo que será utilizado em seu código. Para compreender melhor, faremos alguns exemplos:

Exemplo 1: *typedef int inteiro*

Se criarmos esse tipo em nosso código, estaremos renomeando o tipo *int* com o nome *inteiro*, assim, TODAS as características que existem em *int* poderão ser aplicadas no tipo *inteiro*. Veja um código com esse exemplo.

```
1 #include <stdio.h>
2 typedef int inteiro;
3
4 inteiromain() {
5     inteiro x;
6     printf("\nDigite um número inteiro: " );
7     scanf("%i", &x );
8
9     printf("\nNúmero = %i", x );
10    return 0;
11 }
```

Observe que, em todos os lugares onde o tipo *inteiro* foi utilizado, as características aplicadas são as mesmas do tipo *int*, já conhecido.

Você deve ter pensado assim, mas para que eu faria isso? Concordo, mas observe que este é apenas um exemplo, para auxiliar seu entendimento, vamos fazer mais dois exemplos “simples” e em breve aplicaremos de forma mais prática. Só compreenda a utilização para não se confundir. ok?



Exemplo 2: `typedef int inteto[5]`

Se criarmos esse tipo em nosso código, estaremos informando que sempre que declararmos uma variável com o nome `inteto`, ela será um vetor de inteiros com 5 posições levando consigo TODAS essas características. Veja um código com esse exemplo.

```
1 #include<stdio.h>
2 typedef int inteto[5];
3
4 int main() {
5     inteto x;
6     int i;
7     for (i = 0; i < 5; i++) {
8         x[i] = i * 10;
9         printf("\nNúmero = %i", x[i]);
10    }
11    return 0;
12 }
```

Observe que, neste código, ao declarar a variável `x`, ela se “tornou” um vetor de `int` com 5 posições, afinal, isto foi predefinido na declaração do tipo, pelo `typedef`.

Exemplo 3: `typedef int* inteto`

Se criarmos esse tipo em nosso código, estaremos informando que sempre que declararmos uma variável com o nome `inteto`, ela será um ponteiro de inteiros levando consigo TODAS as suas características. Veja um código com esse exemplo:

```
1 #include<stdio.h>
2 typedef int *inteto;
3
4 int main() {
5     inteto x;
6     int i = 7;
7     x = &i;
8     *x = 89;
9     printf("\nNúmero = %i", *x);
10    return 0;
11 }
```

Observe que, neste código, ao declarar a variável `x`, ela se “tornou” um ponteiro de `int`, e por isso foi possível armazenar o endereço de memória da variável `i`.

Agora que você compreendeu a utilização do `typedef`, chegou a hora de fazer uma estrutura com as abstrações que falamos.



Estrutura (struct)

A partir de agora conheceremos a forma de aplicar, no código, o conceito de endereço, citado no início deste texto.

Na linguagem C, isto será feito com o conceito de estrutura (*struct*), as estruturas permitem que diversos dados sejam agrupados, criando assim um item único com diversos elementos.

OBSERVAÇÃO

O exemplo anterior é ótimo para nossa compreensão, por isso, utilizaremos ele e aplicaremos na linguagem.



A sintaxe para criação de estruturas em C é:

```
struct <nome> {
    tipo membro01;
    tipo membro02;
    ...
    tipo membroN;
}
```

O nome de uma *struct* segue a mesma regra para a declaração de uma variável, definido por você e utilizando bom senso.

Os tipos dos membros são os tipos primitivos, já conhecidos: *char*, *int*, *float*, *double*; pode-se também utilizar uma *struct* como tipo definido; além de ponteiros de qualquer tipo (primitivo ou não).

Aplicando essa sintaxe nosso exemplo ficará desta forma:

```
struct dadosEndereco{
    char rua[40];
    int numero;
    char complemento[30];
    char bairro[20];
    char cidade[20];
    char estado[2];
    int cep;
}
```



Agora vamos aplicar e utilizar um membro desta estrutura:

```
1 #include <stdio.h>
2 struct dadosEndereco{
3     char rua[40];
4     int numero;
5     char complemento[30];
6     char bairro[20];
7     char cidade[20];
8     char estado[2];
9     int cep;
10 };
11
12 int main() {
13     printf("\nInício do código.");
14     struct dadosEndereco end; //A variável end é do tipo struct dadosEndereco
15     //Atribuir valor a um membro da estrutura
16     end.numero = 12; //O membro numero da variável foi preenchido com 12
17     printf("\nNúmero = %i", end.numero);
18
19     //Utilizar comando de entrada com um membro da estrutura
20     printf("\nDigite o CEP: ");
21     scanf("%i", &end.cep);
22     printf("\nCEP = %i", end.cep);
23
24     return 0;
25 }
```

No código podemos observar que a estrutura foi declarada de forma global, portanto, poderá ser utilizada em qualquer função. Ela é apenas uma abstração do que será declarado caso utilizemos este tipo, por isso, para realmente utilizar precisamos declarar, pode ser variável, retorno de função, etc.

ATENÇÃO

Para acessar qualquer membro é importante se lembrar que isto NÃO é uma variável comum, pertence a uma estrutura, por isso, você precisará chamar a variável representada e só depois acessar o membro. No código, como nossa variável tem o nome de *end*, antes de chamar qualquer membro temos que acessá-la primeiro, como por exemplo, o que foi feito na linha 15, *end.numero*;

Outra coisa muito importante é que, após acessar o membro, o tipo final será referente a ele, por exemplo, a nossa variável *end* é do tipo *struct dadosEndereco*, mas o membro *cep* é do tipo *int*, por isso, ao utilizarmos os comandos de entrada e saída formatamos para *int*.



Como você já conhece o comando *typedef* deve ter se perguntado “Porque não aplicar o *typedef* aqui?” ótima pergunta, a verdade é que o ideal é realmente fazer essa criação com o *typedef*, veja como ficaria o código!



```

1 #include<stdio.h>
2 typedef struct dadosEnderecadosEndereco; //Posso utilizar o mesmo nome
3 struct dadosEndereco{
4     char rua[40];
5     int numero;
6     char complemento[30];
7     char bairro[20];
8     char cidade[20];
9     char estado[2];
10    int cep;
11 };
12
13 int main() {
14     printf("\nInício do código.");
15     dadosEnderecoend;//A variável end é do tipo dadosEndereco
16     //Atribuir valor a um membro da estrutura
17     end.numero = 12; //O membro numero da variável foi preenchido com 12
18     printf("\nNúmero = %i", end.numero );
19
20     //Utilizar comando de entrada com um membro da estrutura
21     printf("\nDigite o CEP: ");
22     scanf("%i", &end.cep );
23     printf("\nCEP = %i", end.cep);
24
25     return 0;
26 }
```

Agora, além dos tipos básicos, já utilizados por nós, também podemos utilizar um tipo denominado *dadosEndereco*, que possui características definidas para este código.

E como este é um tipo, você deve ter se perguntado.... **Posso declarar uma estrutura dentro de outra?**

Minha resposta é sim.... você pode declarar uma estrutura dentro de outra, afinal, uma vez definido um tipo, ele poderá ser utilizado como você quiser... pode declarar variável, utilizar como retorno de função, ponteiro, parâmetro, etc.



REFLITA

Vamos pensar em um exemplo de utilização de uma estrutura dentro de outra. Imagine que você precisará armazenar alguns dados de cliente, dentre eles o endereço.

Os dados que pensei foram: nome, idade, salário, gênero e endereço (completo).

Bem, vamos pensar assim, um tipo endereço, com os dados definidos, eu já tenho, então pretendo utilizá-lo, agora falta modelar o cliente para utilizar.



```

1 #include<stdio.h>
2 typedef struct dadosEnderecadosEndereco; //Posso utilizar o mesmo nome da struct
3 struct dadosEndereco{
4     char rua[40];
5     int numero;
6     char complemento[30];
7     char bairro[20];
8     char cidade[20];
9     char estado[2];
10    int cep;
11 };
12
13 typedef struct dadosCliente cliente; //Posso utilizar nomes diferentes
14 struct dadosCliente{
15     char nome[30];
16     int idade;
17     float salario;
18     char genero;
19     dadosEnderecoend; //Tipo previamente declarado neste código
20 };
21
22 int main() {
23     printf("\nInício do código.");
24     cliente cli[2];//O vetor cli é do tipo cliente
25     int i;
26
27     //Preencher alguns itens do cliente, para exemplificar.
28     for (i = 1; i <= 2; i++) {
29         printf("\nDigite o salário do cliente: ");
30         scanf("%f", &cli[i].salario); //Recebe a digitação do salário
31
32         printf("\nDigite a cidade do cliente");
33         gets(cli[i].end.cidade);
34     }
35
36     return 0;
37 }

```



União (Union)

A criação de uma união é muito parecida com a criação de uma struct, porém, o intuito de criarmos uma union é na verdade aproveitar o espaço de memória único.

Bem, imagine que você possui três variáveis em seu código, mas tem certeza de que nunca utilizará as três ao mesmo tempo, sempre que utilizar uma a outra estará disponível, e por aí vai. Bem, essa é uma ótima oportunidade para que você utilize uma union, pois ela permite que você reúna dados heterogêneos, com economia de memória.

Vamos ver um exemplo e compreender melhor:

```
1 union valores{  
2     int valInt;  
3     double valDouble;  
4     char valChar;  
5 }
```

Na union declarada temos três tipos distintos, um *int*, outro *double* e outro *char*, para a utilização desta união, o espaço de memória será compartilhado, por isso, você só poderá utilizar um membro por vez, ou seja, quando o seu código utilizar a variável *int*, os outros estarão “desabilitados”, sem utilização ou conteúdo, e quando seu código mudar e utilizar a variável *char*, o valor que estava na variável *int* será perdido e apenas a variável *char* estará disponível. Entendeu?

```
1 int main(){  
2     union valores val;  
3     val.valDouble = 15.87;  
4     printf("%lf - %c - %i", val.valDouble, val.valChar, val.valInt);  
5     //Neste item apenas a variável valDouble terá valor, as demais estarão indisponíveis  
6     val.valInt = 15;  
7     printf("%lf - %c - %i", val.valDouble, val.valChar, val.valInt);  
8     //Neste item apenas a variável valInt terá valor, as demais estarão indisponíveis  
9     val.valChar = 'A';  
10    printf("%lf - %c - %i", val.valDouble, val.valChar, val.valInt);  
11    //Neste item apenas a variável valChar terá valor, as demais estarão indisponíveis  
12 }
```

Ah, se você testar, perceberá que a memória separada para este tipo de estrutura considera a maior quantidade de bytes necessária, por exemplo, se em nossa arquitetura uma variável *int* gasta 4 bytes, uma variável *double* gasta 8 bytes e uma variável *char* gasta 1 byte, o tipo *union valores* separará 8 bytes para inserir todos eles, pois ele considera que você armazenará, no máximo, 8 bytes em cada utilização.

E antes que eu me esqueça, aqui você também pode utilizar o *typedef*, e assim criar um novo tipo sobre a *union* declarada.

```
1 typedef union valores valUnion;
```

Utilizando o *typedef*, qualquer variável declarada com essa característica será do tipo *valUnion*, e este será um tipo reconhecido pelo código.



Enumeração (Enum)

A enumeração permite reunir valores inteiros (constantes) em um único item, ficando assim mais organizado e fácil de utilizar, ao invés de termos várias variáveis, inteiras teremos apenas uma enumeração com os valores possíveis.

O exemplo mais clássico é o do booleano, afinal, não existe este tipo como primitivo em C, então, para criá-lo podemos pensar o seguinte:

Começaremos declarando um enum com os valores possíveis *TRUE* e *FALSE*.

```
1 enum bool{  
2     FALSE,  
3     TRUE  
4 }
```

Como eu disse, aqui reunimos valores inteiros, por default ele iniciará com o valor (zero), então *FALSE* vale 0 e *TRUE* vale 1, mas, se for necessário, você pode utilizar o operador de atribuição e iniciar.

```
1 enum bool{  
2     FALSE = 0,  
3     TRUE  
4 }
```

Concordo que aqui ficou redundante, mas é só para exemplificar. Neste caso “forçamos” para que o membro *FALSE* comece com o valor 0 e os demais seguirão sempre + 1, então, o valor *TRUE* vale 1.

Sua aplicação no código seria possível acessando desta forma:

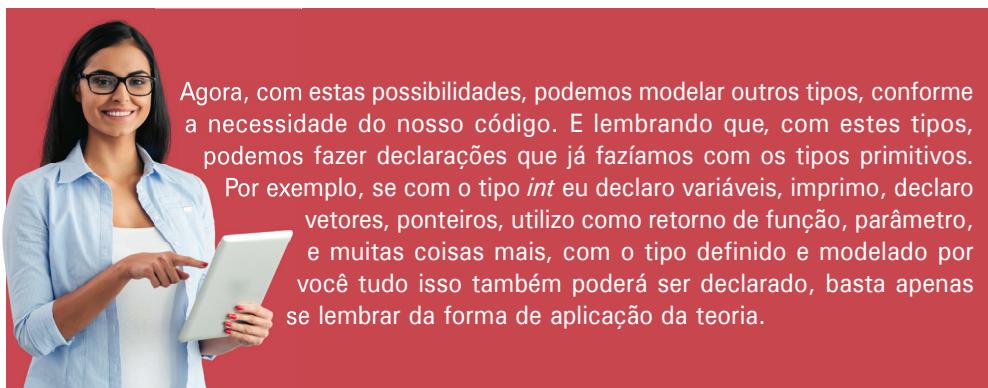
```
1 int main(){  
2     enum boolboolean;  
3     printf("%i - %i", FALSE, TRUE);  
4 }
```

Aqui também você pode aplicar o *typedef* para criar “novos” tipos dentro do código.

```
1 typedef enum boolboolean;
```

Com isso, qualquer variável declarada com essa característica será do tipo *boolean*, e este será um tipo reconhecido pelo código.

Além disto, *structs*, *union* e *enum* podem ser passadas como parâmetros e/ou declaradas como ponteiros.



Síntese

Assim finalizamos o conteúdo sobre tipos abstratos de dados.

Você descobriu que é possível modelar tipos conforme a necessidade do seu código, fazendo com que todas os dados relevantes fiquem em apenas um local, com acesso direto ao item que precisar.

Viu que as modelagens possíveis são: estruturas, declaradas como structs, que nos permite reunir diversos tipos (primitivos ou não), modelados no mesmo item; união, declarados como union, que nos permite aproveitar melhor a memória, quando os itens não forem utilizados ao mesmo tempo; e, por fim, a enumeração, declaradas como enum, que nos permite enumerar itens pré-existentes.

Também percebeu que utilizar um tipo abstrato de dado, criado e mantido por você, segue as mesmas premissas de um tipo primitivo, já utilizado em outros códigos.

Agora, seus códigos ficarão mais bem elaborados e completos.

Até mais!

Referências

CELES, Waldemar, CERQUEIRA, Renato, RANGEL, José Lucas. **Introdução a estruturas de dados:** com técnicas de programação em C. Rio de Janeiro: Campus, 2004. 294 p

BACKES, André. **Linguagem C :** completa e descomplicada. 2. ed. Rio de Janeiro: Elsevier, 2019.

DAMAS, Luis. **Linguagem C.** 10. ed. Rio de Janeiro: Ltc, 2016.

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C.** São Paulo: Pearson Prentice Hall, 2008. 407 p.

REESE, Richard. **Understanding and using C pointers.** Sebastopol: O'Reilly, 2008. 407 p.