LINGUAGEM DE PROGRAMAÇÃO

AULA 3

Prof. Sandro de Araujo

CONVERSA INICIAL

Essa aula teve como base o livro *Treinamento em Linguagem C*, de Viviane Victorine Mizrahi. Em caso de dúvidas ou aprofundamento consulte-o.

A aula apresenta a seguinte estrutura de conteúdo:

1. *Struct;*
2. *Union;*
3. *Enum;*
4. *Typedef;*
5. *Typedef* e *struct.*

O objetivo dessa aula é conhecer os principais conceitos de *struct*, *union*, *enum* e *typedef* na linguagem de programação C, e representá-los facilmente em diversos algoritmos para resolver problemas computacionais.

TEMA 1 – *STRUCT*

Uma *struct* pode ser compreendida como um conjunto de variáveis referenciadas pelo mesmo nome, sendo que cada uma delas pode ter um mesmo tipo de dado, ou então vários.

A ideia básica por trás da uma *struct* é criar uma variável que contenha várias outras variáveis, ou seja, estamos criando uma variável que contém dentro de si outras variáveis.

Na linguagem de programação C podemos declarar tipos de variáveis como:

- Tipos básicos: char, int, float, double;
  - ✓ Exemplo: int x; float y;
- Tipos compostos homogêneos: array;
  - ✓ Exemplo: int x[5]; char nome[25].

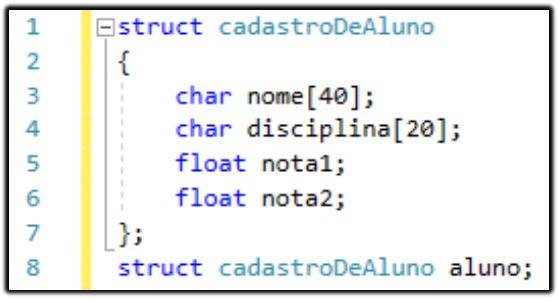
Além dos tipos de dados mencionados acima a linguagem de programação C permite a criação dos nossos próprios tipos de variáveis e um desses tipos é a estrutura ou *struct*.

A *struct* segue a seguinte sintaxe:

```
struct <nome_da_struct>
{
                <tipo 1> e <variável 1>;
                <tipo 2> e <variável 2>;
                <tipo 3> e <variável 3>;
                ...
                <tipo n> e <variável n>;

}
struct < nome_da_struct > <nome_variavel>;
```

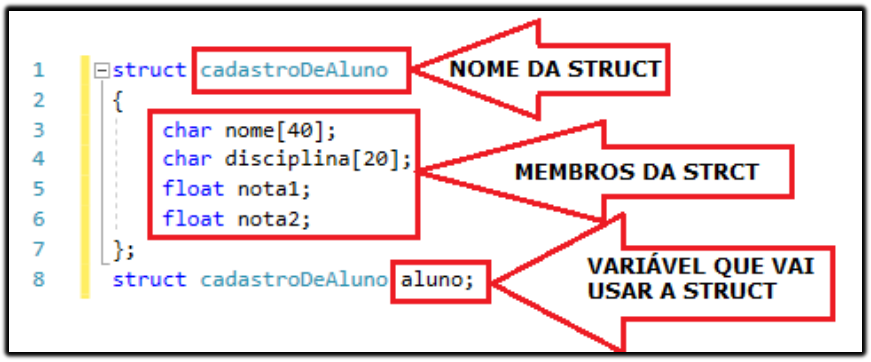
Exemplo de declaração de uma *struct*:

Figura 1 – Declaração de uma *struct*



A Figura 2 mostra o algoritmo acima de forma detalhada.

Figura 2 – Declaração de uma struct



No exemplo acima temos o nome da *struct* (cadastroDeAluno), os membros que compõem a estrutura (char nome[40], char disciplina[20], float nota1 e float nota2) e a variável que vai usar a *struct* (aluno). Dizemos que a variável **aluno** é do tipo **cadastroDeAluno** (struct cadastroDeAluno aluno). A Figura 3 traz um algoritmo com um exemplo de uma *struct*.

Figura 3 – Algoritmo com *struct*

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

int main() {

    printf("\n CADATRO DE CLIENTE\n\n");

    struct ficha_do_cliente     /*Criando a struct */
    {
        char nome[50];
        char rua[50];
        int telefone[11];
        char email[40];
    }; struct ficha_do_cliente cliente; /*Criando a variável aluno que
                                        será do tipo struct ficha_de_aluno */

    printf("\n************** Lendo dados do Cliente **************\n\n\n");

    printf("Digite o nome do Cliente: ");
    fflush(stdin);
    fgets(cliente.nome, 50, stdin);

    /*usaremos o comando fgets() para ler strings, no caso o nome do aluno e
    a disciplina fgets (variável,   tamanho da string, entrada) como estamos
    lendo do teclado a entrada é stdin (entrada padrão),
    porém em outro caso, a entrada também poderia ser um arquivo */

    printf("Rua: ");
    fflush(stdin);
    fgets(cliente.rua, 30, stdin);

    printf("Telefone: ");
    scanf_s("%d", &cliente.telefone);

    printf("E-mail: ");
    fflush(stdin);
    fgets(cliente.email, 30, stdin);

    printf("\n*************** Imprimindo os dados da struct  ************\n");
    printf("****************** Nome....: %s", cliente.nome);
    printf("****************** Rua.....: %s", cliente.rua);
    printf("****************** Telefone: %d", cliente.telefone);
    printf("\n****************** E-mail..: %s", cliente.email);
    printf("\n\n");
    getch();
    return 0;
}
```

A Figura 4 mostra a saída do algoritmo acima após a sua execução:

Figura 4 – Saída do algoritmo

TEMA 2 – *UNION*

Com a *union* pode-se criar variáveis capazes de suportar dados diferentes, alocados no mesmo espaço de memória, em momentos diferentes. Isto é, a *union* permite que um conjunto de variáveis compartilhem o mesmo espaço na memória

Declara-se uma *union* de forma muito semelhante à uma *struct*; estas só se diferem no aspecto da *struct* ser alocada com espaço suficiente para todos os objetos, e o *union* só aloca espaço para o maior objeto que o compõe. Esse espaço alocado é suficiente para armazenar o maior dos seus membros.

Portanto, uma *union* define um conjunto de membros que serão armazenados numa porção compartilhada da memória, isto é, apenas um membro será armazenado de cada vez.

A sintaxe da *union* é similar à da *struct* , conforme mostrado a seguir:

```
union <nome_da_union>
{
        <tipo 1> e <variável 1>;
        <tipo 2> e <variável 2>;
        <tipo 3> e <variável 3>;
        ...
        <tipo n> e <variável n>;
};
```

A declaração começa com a palavra *union* seguida do identificador da união. A primeira linha da declaração indica ao compilador que este é um novo

tipo de dados para ser usado em outras partes do programa. Após as chaves, declara-se as variáveis que vão compor a *union*.

No exemplo da Figura 5 vamos trabalhar com números, porém não sabemos se os dados serão do tipo **int** ou **float**. Portanto, declara-se um novo tipo de dados, que iremos chamar de *numeroFlex*, capaz de armazenar valores de um dos dois tipos:

Figura 5 – Declaração de uma *union*

```
1  union numeroFlex {
2      int num1;
3      float num2;
4  };
```

Ao acessar uma variável de tipo *union*, precisamos também indicar qual a variável que desejamos acessar. A sintaxe é a mesma que aquela para o acesso de membros em uma *struct*. Conforme exemplo na Figura 6 que vai declarar e atribuir um valor inteiro à variável **num**:

Figura 6 – Declaração e atribuição de uma *union*

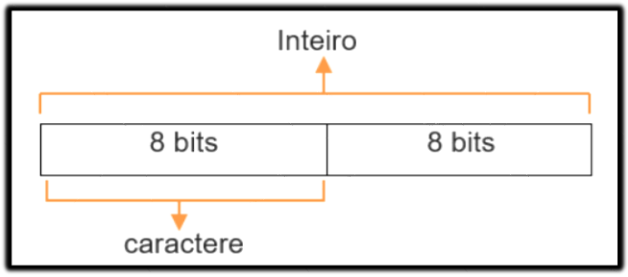
```
1  union numeroFlex num;
2  num.num1 = 10;
```

Para atribuir um valor real à variável **num** usou-se a seguinte instrução:

num.num2 = 4.0;

A particularidade da *union* está no fato de que as variáveis num.num1 e num.num2 vão compartilhar a mesma posição na memória, e a atribuição de um valor fracionário sobrescreve o valor inteiro e vice-versa. Portanto, não há uma forma de "consultar" uma *union* e sobre qual tipo ela está armazenando em um dado momento. Tampouco existe uma forma para o programa decidir automaticamente a que variável da união ele deve atribuir um valor. Este controle fica por conta do programador. A Figura 7 ilustra o compartilhamento de espaço na memória.

Figura 7 – Compartilhamento da memória entre variáveis



A Figura 7 representa que se aloca a quantidade de armazenamento ocupada pelo maior membro da union. Com isso, a memória economiza espaço de armazenamento.

No próximo exemplo, na Figura 8, vamos trabalhar com um inteiro e um caractere. Para esse algoritmo declara-se um novo tipo de dado que iremos chamar de totalFlex, capaz de armazenar valores de um dos dois tipos (inteiro e caractere).

Figura 8 – Union com um inteiro e um caractere

```
1  union totalFlex {
2      int numero;
3      char letra;
4  };
```

Uma *union* torna o código fonte um pouco confuso e, portanto, deve ser utilizada em momentos em que as variáveis não sejam executadas no mesmo momento.

Para um melhor entendimento vejamos o algoritmo em que ele imprime o tamanho de uma *union* e *struct*, ambas com os mesmos tipos de variáveis (Figura 9).

Figura 9 – Compartilhamento da memória entre variáveis

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

union ex_union {

    int inteiro_1;
    char caractere_1;
    float decimal_1;

};

struct ex_struct {

    int inteiro_2;
    char caractere_2;
    float decimal_2;

};

int main()
{
    printf("Tamanho da Union: %d\n", sizeof(union ex_union));
    printf("Tamanho da Struct: %d\n", sizeof(struct ex_struct));

    system("pause");
    return 0;
}
```

A Figura 10 mostra a saída do algoritmo acima após a sua execução:

Figura 10 – Compartilhamento da memória entre variáveis

```
Tamanho da Union: 4
Tamanho da Struct: 12

Process returned 0 (0x0)   execution time : 0.115 s
Press any key to continue.
```

No exemplo da Figura 10 podemos verificar que a *union* ocupou um terço do espaço ocupado pela *struct* com o método de compartilhamento de espaço.

TEMA 3 – *ENUM*

Enum ou *enumeração*, é um tipo de dado definido pelo usuário, com o uso de uma lista de identificadores. Os identificadores podem ser vistos como uma lista de constantes, onde cada constante tem um nome significativo.

A *enum* segue a seguinte sintaxe:

enum <nome_da_enum>{lista_de_identificadores};

Assim como em um vetor, uma estrutura *enum* também começa com o valor zero (0). Para adequar uma *enum* em um conjunto de constantes que referencie os meses do ano instanciamos o valor um (1) ao mês de janeiro, conforme o exemplo mostrado na Figura 11.

Figura 11 – Declaração e atribuição de uma enum

```
1  enum meses {
2      janeiro = 1, fevereiro, marco, abril, maio, junho, julho,
3      agosto, setembro, outubro, novembro, dezembro
4  };
```

A Figura 12 apresenta um exemplo com os dias da semana. Para esse exemplo não inicializamos nenhum membro da *enum*.

Figura 12 – Compartilhamento da memória entre variáveis

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  enum semana { domingo, segunda, terca, quarta, quinta, sexta, sabado };
4
5  int main() {
6
7      enum semana x; // Declaração de uma variável enum
8      x = quarta; //x recebe o valor 3
9      printf("Valor de quarta = %d\n\n", x); //imprime o identificador quarta
10
11     system("pause");
12     return 0;
13 }
```

A Figura 13 mostra a saída do algoritmo acima após a sua execução:

Figura 13 – Compartilhamento da memória entre variáveis

```
Valor de quarta = 3

Pressione qualquer tecla para continuar. . .
```

O identificador quarta apresentou o valor três (3). Para resolver esse problema devemos definir no **enum semana** uma enumeração em que é atribuído um identificador para a primeira constante com valor um (1); a *enum*, portanto, vai incrementar com o valor um (1) de forma sequencial os demais identificadores até o fim da lista. Com isso, a lista terá os valores de um a sete para os dias da semana.

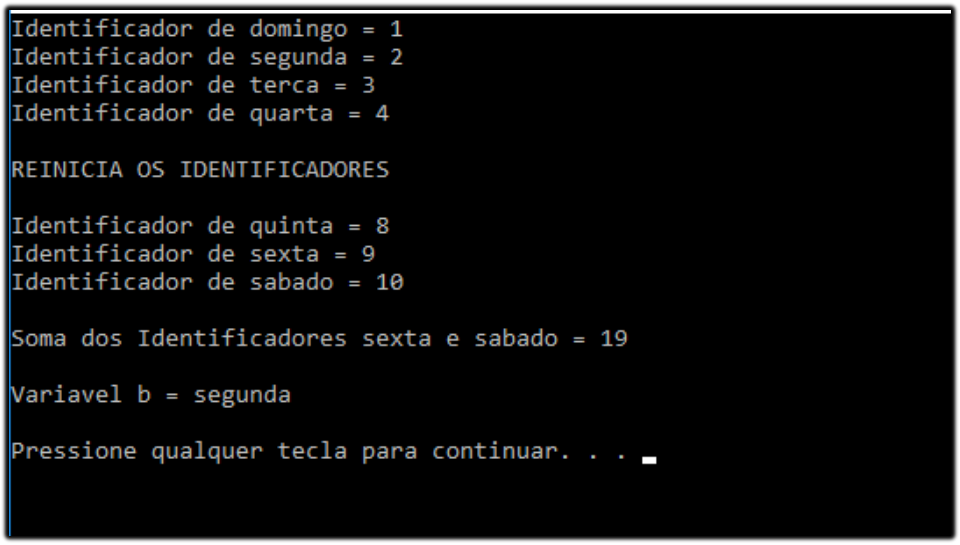
Para melhor entendimento vejamos um algoritmo, na Figura 14, que vai executar o mesmo exemplo com dois identificadores instanciados.

Figura 14 – Operações com identificadores

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

enum semana {
    domingo = 1, segunda, terca, quarta, quinta = 8, sexta, sabado
};

int main() {

    enum semana a, b, c, d, x, y, z, i;

    //atribuição de um identificador para uma variável
    a = domingo;
    b = segunda;
    c = terca;
    d = quarta;
    x = quinta;
    y = sexta;
    z = sabado;

    i = y + z;  //soma de dois identificadores

    printf("Identificador de domingo = %d\n", a);
    printf("Identificador de segunda = %d\n", b);
    printf("Identificador de terca = %d\n", c);
    printf("Identificador de quarta = %d\n", d);

    printf("\nREINICIA OS IDENTIFICADORES\n\n");

    printf("Identificador de quinta = %d\n", x);
    printf("Identificador de sexta = %d\n", y);
    printf("Identificador de sabado = %d\n", z);

    printf("\nSoma dos Identificadores sexta e sabado = %d\n\n", i);

    //comparação com identificador, se for true executa a instrução
    if (b == segunda)
        printf("Identificador b = a segunda\n\n");

    system("pause");
    return 0;
}
```

A Figura 15 mostra a saída do algoritmo acima após a sua execução.

Figura 15 – Operações com identificadores



O exemplo da Figura 9 – o identificador quinta – também foi instanciado. Repare ainda que na Figura 10 o identificador sexta seguiu a sequência, apresentando o valor nove (9).

Podemos também atribuir valores da tabela ASCII para enumeração, conforme mostrado no algoritmo da Figura 16.

Figura 16 – Escapes com identificadores

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4   enum escapes { backspace = '\b', nova_linha = '\n', tabulacao_h = '\t' };
5
6
7  int main()
8  {
9      enum escapes esc = nova_linha;
10     printf("Testando %c%c%c de %c%c%c", esc, esc, esc, esc, esc, esc);
11
12     esc = tabulacao_h;
13     printf("Testando%cTestando%cTestando\n\n", esc, esc);
14
15     system("pause");
16     return 0;
17 }
```

A Figura 17 mostra a saída do algoritmo acima após a sua execução:

Figura 17 – Escapes com identificadores

```
Testando

 de

Testando        Testando        Testando
Pressione qualquer tecla para continuar. . .
```

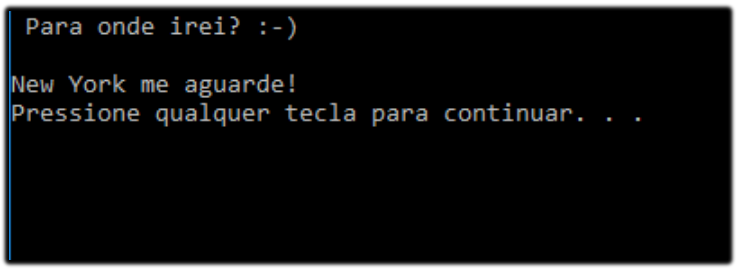
A Figura 18 traz outro exemplo com estrutura *enum*.

Figura 18 – Operações com identificadores

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

void mostrarRes(int pais);

//Aqui os valores Italia = 4 e USA = 5
enum { FRANÇA = 3, ITALIA, USA };

int main()
{
    int n = USA;
    mostrarRes(n);

    system("pause");
    return 0;
}

void mostrarRes(int pais)
{
    printf(" Para onde irei? :-) \n \n");

    switch (pais)
    {
    case USA: printf("New York me aguarde!\n");
        break;
    case FRANÇA: printf("Paris cidade apaixonante.\n");
        break;
    case ITALIA: printf("Florença teem a melhor pizza.\n");
        break;
    default: printf("Preciso viajar.\n");

    }
}
```

A Figura 19 mostra a saída do algoritmo acima após a sua execução:

Figura 19 – Operações com identificadores



TEMA 4 – *TYPEDEF*

O comando *typedef* é usado para criar "sinônimo" ou um "*alias*" para tipos de dados existentes. Na prática podemos dizer que estamos renomeando um tipo de dados. Essa renomeação de tipos facilita a organização e o entendimento do código. Sintaxe:

typedef <nome do tipo de dado> <novo nome>;

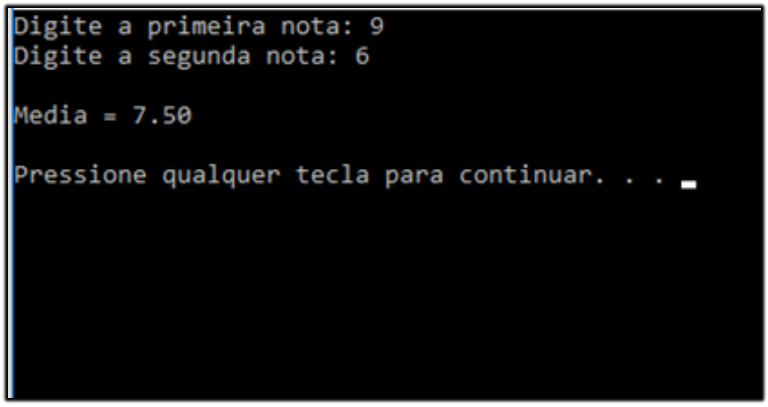
É importante ressaltar que o comando *typedef* não cria um novo tipo. Ele apenas permite que um tipo existente seja denominado de uma forma diferente, de acordo com a especificação desejada pelo programador, conforme mostrado no algoritmo da Figura 20:

Figura 20 – Renomeação de tipo de variável com *typedef*

```c
#include <stdio.h>
#include <conio.h>

int main()
{
    //redefinição do tipo float para o tipo prova
    typedef float prova;

    // variáveis usando o tipo prova
    prova nota1, nota2, media;

    printf("Digite a primeira nota: ");
    scanf_s("%f", &nota1);

    printf("Digite a segunda nota: ");
    scanf_s("%f", &nota2);

    media = (nota1 + nota2) / 2;

    printf("Media = %.2f\n", media);

    system("pause");
    return 0;
}
```

A Figura 21 mostra a saída do algoritmo acima após a sua execução:

Figura 21 – Renomeação de tipo de variável com *typedef*



TEMA 5 – *TYPEDEF* E *STRUCT*

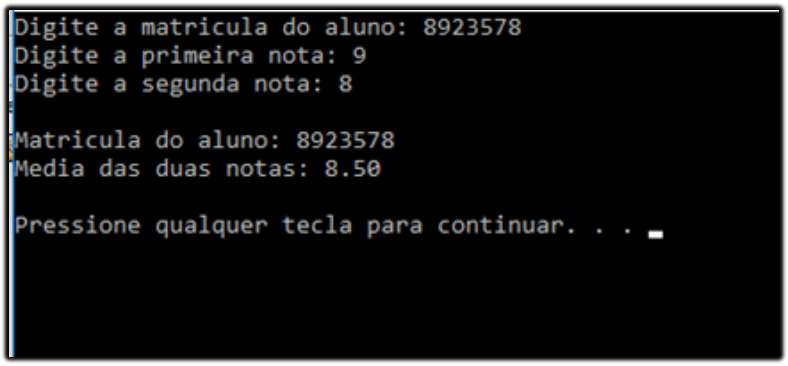
É muito frequente o uso de *typedef* para criar apelidos a fim de tornar os nomes mais curtos, desta forma podemos representar uma estrutura usando apenas seu sinônimo (Figura 22).

Figura 22 – *Typedef* e *struct*

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

typedef float prova; //redefinindo float
typedef int RU; //redefinindo int

struct notasAluno
{
    RU matricula; //apelidos dentro da struct
    prova nota1;
    prova nota2;
}; typedef struct notasAluno n_aluno; //criando um apelido para a struct

int main()
{
    n_aluno aluno; // Não é mais necessário escrever "struct n_aluno"
    prova media = 0;

    printf("Digite a matricula do aluno: ");
    scanf_s("%d", &aluno.matricula);

    printf("Digite a primeira nota: ");
    scanf_s("%f", &aluno.nota1);

    printf("Digite a segunda nota: ");
    scanf_s("%f", &aluno.nota2);

    media = (aluno.nota1 + aluno.nota2) / 2;

    printf("\nMatricula do aluno: %d\n", aluno.matricula);
    printf("Media das duas notas: %.2f\n\n", media);

    system("pause");
    return 0;
}
```

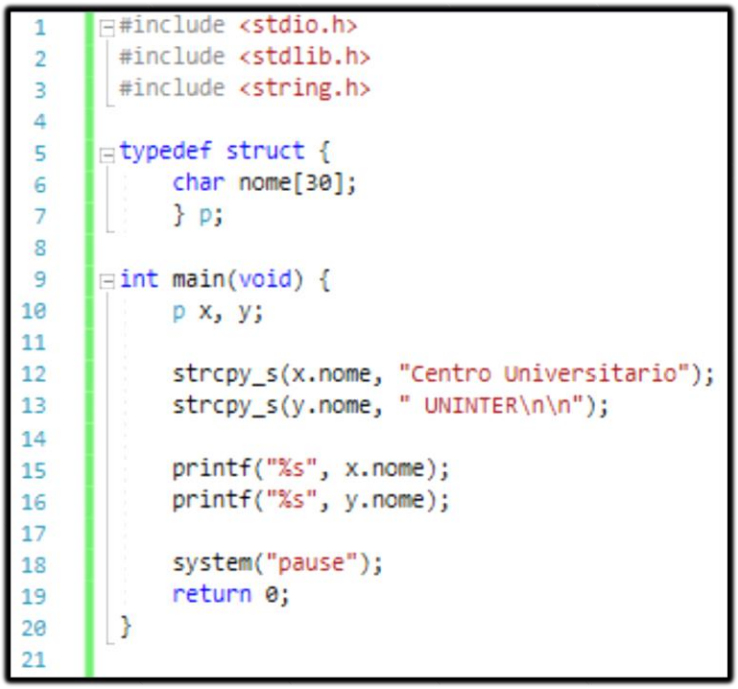
A Figura 23 mostra a saída do algoritmo acima após a sua execução:

Figura 23 – *Typedef* e *struct*



A Figura 24 traz outro exemplo de *typedef* com *struct*.

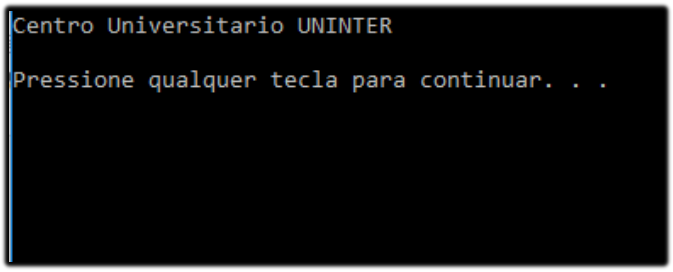
Figura 24 – *Typedef* e *struct*



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct {
6      char nome[30];
7      } p;
8
9  int main(void) {
10     p x, y;
11
12     strcpy_s(x.nome, "Centro Universitario");
13     strcpy_s(y.nome, " UNINTER\n\n");
14
15     printf("%s", x.nome);
16     printf("%s", y.nome);
17
18     system("pause");
19     return 0;
20 }
21
```

A Figura 25 mostra a saída do algoritmo acima após a sua execução:

Figura 25 – *Typedef* e *struct*



FINALIZANDO

Nesta aula aprendemos os principais conceitos referente a *struct*, *union*, *enum* e *typedef* na linguagem de programação C, e também como representá-los facilmente em algoritmos nas diferentes estruturas para resolver problemas computacionais.

Aproveite a disciplina e bons estudos!

REFERÊNCIAS

MIZRAHI, V. V. **Treinamento em Linguagem C**. [S.l.]: Edição da autora, 2008.