



# Algoritmos e Programação de Computadores

Tipos definidos pelo programador

Prof. Lucas Boaventura  
lucas.boaventura@unb.br





# Introdução

- Até agora, trabalhamos com tipos básicos de dados no nosso curso:
- int, char, float, double.
- Esses dados podem ser agrupados em vetores e/ou matrizes quando são homogêneos, ou seja, são iguais
- Mas o que fazer quando deseja-se agrupar dados de tipos diferentes?





# Introdução

- Por exemplo, um sistema de notas da matéria de APC
- Nele, desejamos armazenar o nome dos alunos (string), a matrícula dos alunos (int, string) e as notas das provas (vetor de float/double)





# Introdução

- Uma alternativa simples, seria declarar vetores de variáveis para cada tipo a ser armazenado:

```
char nome_alunos[100][100];  
int matricula_alunos[100];  
float notas[100][3];
```

- Mas, para um sistema grande, o sistema deve ser dividido em funções:

```
void listar(char nome_alunos[][], int matricula_alunos[], float notas[][3])
```

```
void aprovar(char nome_alunos[][], int matricula_alunos[], float notas[][3])
```

~



# Introdução

- Um sistema grande iria precisar tipicamente de centenas de funções
- Todas as funções iriam receber diversos argumentos
- O que aconteceria se tivéssemos que mudar o tipo da nota (de float para double), ou a matrícula dos alunos (de int para string)?
- Todas as funções seriam reescritas





# Introdução

- Esse problema acontece pois não definimos um tipo de dados para um “aluno”
- O que nós desejamos é declarar um tipo de variável aluno, que agrupa todas as informações de um aluno:

```
aluno lista_de_alunos[100];
```

```
void listar(aluno lista_de_alunos[]);
```

```
void aprovar(aluno lista_de_alunos[]);
```





# Introdução

- Para atingir esse objetivo, é permitido que o programador defina ele mesmo tipo de dados
- Na aula de hoje, iremos aprender diferentes tipos:
  - Estruturas (struct);
  - Uniões (union);
  - Enumerações (enum);
  - Definição própria de tipos (typedef).





# Struct

- Uma estrutura é um conjunto de diferentes variáveis definidas e agrupadas por um nome em comum
- Uma estrutura busca definir um tipo de dados que é composto por outros tipos de dados







# Struct

- Em C, define-se uma estrutura utilizando a palavra-chave reservada **struct**

```
struct minha_struct {  
    tipoA campoA;  
    tipoB campoB;  
    tipoC campoC;  
    ...  
};
```





# Struct

- No nosso exemplo anterior, podemos definir aluno como:

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
};
```





- Declarando struct aluno e, dentro de main, criando uma variável chamada joao e um int i

```
#include <stdio.h>

struct aluno {
    char nome[100];
    int matricula;
    float notas[3];
};

int main()
{
    struct aluno joao;
    int i;
    return 0;
}
```





# Struct

- Para acessar os campos de uma struct utiliza-se o operador ponto “.”
- Para utilizá-lo, usa-se o nome da variável do tipo que deseja utilizar, o operador ponto e o campo que deseja-se acessar





# Struct

```
#include <stdio.h>

struct aluno {
    char nome[100];
    int matricula;
    float notas[3];
};

int main()
{
    struct aluno aluno1, aluno2;

    scanf("%d", &aluno1.matricula);
    scanf("%s", aluno1.nome);
    return 0;
}
```





# Struct

- Declarando variável aluno1 do tipo “struct aluno”, atribuindo e imprimindo sua matricula
- Declarando um int i, atribuindo e imprimindo seu valor

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
};  
  
int main()  
{  
    struct aluno aluno1;  
    aluno1.matricula = 1234;  
    printf("%d\n", aluno1.matricula);  
  
    int i;  
    i = 10;  
    printf("%d\n", i);  
    return 0;  
}
```



# Struct

- Além disso, ao se declarar uma struct, também é possível declarar variáveis

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
} aluno1, aluno2;  
  
int main()  
{  
    aluno1.matricula = 1234;  
    printf("%d\n", aluno1.matricula);  
    return 0;  
}
```





# Struct

- Como quase todas as variáveis, ao ser declarada uma struct irá possuir lixo de memória
- Para ser inicializada, pode se utilizar os colchetes, assim como os vetores
- Nesta notação, os campos são inicializados na ordem que são declarados na struct







# Struct

```
#include <stdio.h>

struct aluno {
    char nome[100];
    int matricula;
    float notas[3];
};

int main()
{
    struct aluno maria = { "Maria", 12345, {10.0, 10.0, 10.0}};

    printf("%s\n", maria.nome);
    printf("%d\n", maria.matricula);
    return 0;
}
```





# Struct

- As estruturas podem ser definidas globalmente ou localmente
- Structs locais costumam não ser consideradas uma boa prática de programação, pois não permitem que outros trechos do seu código reutilizem o tipo definido





# Struct

- Cada campo da struct possui um nome, que deve ser único.
- No entanto, structs diferentes podem ter o mesmo campo (assim como funções diferentes podem ter o mesmo nome de variável)

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
};
```

```
struct professor {  
    char nome[100];  
    int matricula;  
    short salário;  
};
```





# Struct

- Uma notação mais moderna é utilizar o nome dos campos na declaração da estrutura
- Essa notação não era permitida nas primeiras versões do C, mas foi incorporada posteriormente para facilitar a manutenção do código
- Em sistemas reais, as estruturas costumam frequentemente adquirir/excluir os campos






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

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
};
```

```
int main()  
{  
    struct aluno maria = {  
        .nome = "Maria",  
        .matricula = 12345,  
        .notas = {10.0, 10.0, 10.0}  
    };  
  
    printf("%s\n", maria.nome);  
    printf("%d\n", maria.matricula);  
    return 0;  
}
```





# Struct

- Esta notação, traz algumas vantagens e é uma boa prática de programação
- Independência da ordem declarada na estrutura
- Melhor manutenção do código, pois permite adicionar e remover campos das estruturas





# Struct

- Também é possível utilizar arrays de estruturas
- Desta forma, pode-se acessar a variável da estrutura utilizando o operador [] e posteriormente usar o operador . para acessar o campo
- `variavel[num].campo = (...)`





# Struct

```
struct aluno {  
    char nome[100];  
    int matricula;  
    float notas[3];  
};  
  
int main()  
{  
    struct aluno alunos[10], meu_aluno;  
  
    scanf("%s", alunos[0].nome);  
    scanf("%f", &meu_aluno.notas[0]);  
    scanf("%f", &alunos[0].notas[0]);  
    return 0;  
}
```







# Struct

- As operações de atribuição são aceitas nas estruturas
- Todos os valores de uma variável “v1” do tipo “estrutura” pode ser copiado para a variável “v2” com uma simples atribuição:
- $v2 = v1;$
- Note que, se a estrutura for muito grande, essa operação pode ser lenta





# Struct

- Isso acontece pois a operação de atribuição da struct passa a consumir muitos bytes, dependendo do tamanho
- A palavra chave “sizeof” pode ser utilizada em structs e tipos básicos de dados para verificar o tamanho da estrutura





# Struct

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

```
struct numeros {  
    int primeiro;  
    int segundo;  
};
```

```
int main()  
{  
    struct numeros a;  
    int i;  
    char ch;  
  
    printf("%d %d %d\n", sizeof(a), sizeof(i), sizeof(ch));  
}
```

- Saída do programa 8 4 1 (processador x86\_64)





# Union

- Uma union é o agrupamento de vários campos e tipos
- A union deve ser utilizada quando soubermos que os campos são exclusivos
- Isto é, ao se utilizar um campo da união, os outros não são necessários





# Union

- A declaração de uma união é muito semelhante a de uma struct
- No entanto, os campos compartilham a mesma região de memória

```
union u {  
    int idade;  
    int peso;  
};
```






# Union

- Em uma struct, os campos são independentes
- Já em uma união, alterar um campo irá alterar o valor de todos os outros campos!
- Por isso devemos utilizar a union para economizar memória quando soubermos que os outros campos não serão utilizados





```
union u {  
    int idade;  
    int peso;  
};
```

```
struct s {  
    int idade;  
    int peso;  
};
```

Saída do programa: 4 8

```
int main()  
{  
    union u a;  
    struct s b;  
  
    printf("%d %d\n", sizeof(a), sizeof(b));  
    return 0;  
}
```



# Enum

- Muitas vezes, precisamos de uma lista de diferentes valores
- Para isso, a linguagem C permite a declaração de uma enumeração com a palavra chave reservada “enum”
- `enum dias_da_semana {Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado};`







# Enum

```
enum dias_da_semana {Domingo, Segunda, Terca,  
                    Quarta, Quinta, Sexta, Sabado};
```

```
int main()  
{  
    printf("%d %d\n", Domingo, Sabado);  
    return 0;  
}
```

Saída do programa: 0 6





# Enum

- É possível declarar variáveis do tipo enumeração

```
enum dias_da_semana {Domingo, Segunda, Terca,  
                    Quarta, Quinta, Sexta, Sabado};
```

```
int main()  
{  
    enum dias_da_semana d;  
  
    d = Terca;  
    printf("%d\n", d);  
    return 0;  
}
```

Saída do programa: 2





# Enum

- As enumerações também permitem que os valores sejam alterados pelo programador na declaração da enumeração

```
enum dias_da_semana {Domingo = 1, Segunda, Terca,  
                    Quarta, Quinta, Sexta, Sabado};
```

```
int main()  
{  
    printf("%d %d %d\n", Domingo, Sexta, Sabado);  
    return 0;  
}
```

Saída do programa 1 6 7





# Enum

```
enum ordinais {Primeiro = 1, Segundo,  
               Decimo = 10, DecimoPrimeiro,  
               Vigesimo = 20, VigesimoPrimeiro};  
  
int main()  
{  
    enum ordinais d = DecimoPrimeiro;  
    printf("%d %d\n", d, VigesimoPrimeiro);  
    if (d == Vigesimo)  
        printf("Igual a 20!\n");  
    return 0;  
}
```

Saída do programa 11 21





# Enum

- As enumerações e uniões também podem ser declaradas no escopo local ou no escopo global
- Novamente, é uma boa prática manter essas declarações em um escopo global, para facilitar a reutilização do código





# Typedef

- O typedef permite que o programador declare um tipo de variável utilizando outro tipo
- Ele é um apelido, um sinônimo para um tipo já existente





# Typedef

```
typedef int inteiro;  
  
int main()  
{  
    int i;  
    inteiro j = 0;  
  
    i = j;  
    return 0;  
}
```





# Typedef

- Apesar de ser aceito na linguagem, prefere-se utilizar variáveis do mesmo tipo
- No código anterior, seria fazer a atribuições de variáveis “inteiro” são utilizadas apenas em outras “inteiro”







# Typedef

- O typedef é comumente utilizado com structs, uniões e enumerações para facilitar a declaração de variáveis sem o uso das palavras-chaves





# Typedef

- Normalmente, se une a declaração da struct com o typedef... Inicialmente declara-se a struct

```
struct campos_de_um_retangulo {  
    int base;  
    int altura;  
};  
  
int main()  
{  
    struct campos_de_um_retangulo ret;  
    return 0;  
}
```





# Typedef

- Depois, faz-se um typedef

```
struct campos_de_um_retangulo {  
    int base;  
    int altura;  
};  
  
typedef struct campos_de_um_retangulo retangulo;  
  
int main()  
{  
    retangulo ret;  
    return 0;  
}
```





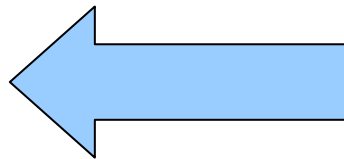
# Typedef

- Depois, faz-se um typedef

```
struct campos_de_um_retangulo {  
    int base;  
    int altura;  
};
```

```
typedef struct campos_de_um_retangulo retangulo;
```

```
int main()  
{  
    retangulo ret;  
    return 0;  
}
```



Não é “struct ...” é  
somente “retangulo”





# Typedef

- Podemos fazer o typedef e struct juntos!!

```
typedef struct campos_de_um_retangulo {  
    int base;  
    int altura;  
} retangulo;
```

```
int main()  
{  
    retangulo ret;  
    return 0;  
}
```





# Typedef

- Para simplificar, a struct chama-se “\_retangulo” e o typedef “retangulo”

```
typedef struct _retangulo {  
    int base;  
    int altura;  
} retangulo;
```

```
int main()  
{  
    retangulo ret;  
    struct _retangulo ret2;  
    return 0;  
}
```





# Obrigado! Dúvidas?

- [lucas.boaventura@unb.br](mailto:lucas.boaventura@unb.br)

