

# Structs

Prof.: Leonardo Tórtoro Pereira

[leonardop@usp.br](mailto:leonardop@usp.br)

# Structs [1]

- Um grupo de elementos de dados agrupados com um mesmo nome é conhecido como *estrutura de dados* ou *struct*
- Esses elementos de dados são chamados de **membros**
- Podem ter diferentes tipos e tamanhos
- Eles são muito úteis para representarmos informações complexas de uma maneira mais organizada!

# Structs [1]

→ Sintaxe de uma *struct*

```
struct nome_tipo {  
tipo_membro1 nome_membro1;  
tipo_membro2 nome_membro3;  
tipo_membro3 nome_membro3;  
.  
.  
} nomes_variaveis;
```

# Structs [1]

- *nome\_tipo* é o nome do tipo da *struct*
- *nomes\_variaveis* pode ser um conjunto de identificadores válidos para as variáveis que pertencem ao tipo dessa *struct*
- Entre chaves estão a lista de membros de dados, cada um com seu tipo e identificador válido como nome.

# Structs [1]

→ Exemplo:

```
struct product {  
    int weight;  
    double price;  
} apple, banana, melon;
```

# Structs [1]

- Assim declaramos uma *struct* de nome *product*
- A *struct* possui dois membros
  - ◆ *peso e preço*
  - ◆ Cada um com seu tipo
- Essa declaração cria um novo tipo: *product*
- Foram criadas 3 variáveis desse tipo:
  - ◆ *apple, banana e melon*
- Porém, podemos declarar variáveis de outro jeito:

# Structs [1]

```
struct product {  
    int weight;  
    double price;  
};
```

```
struct product apple;  
struct product banana, melon;
```

# Structs [1]

- Uma vez que *product* é declarado, ele pode ser usado como um tipo comum, desde que seja colocada a palavra-chave *struct* antes do nome da *struct*
- Nota: em C++ é possível declarar variáveis sem o uso da palavra-chave *struct*. Cuidado para não confundir!
  - ◆ Existe um jeito de fazer isso em C, vamos ver mais adiante nesta aula

# Structs [1]

- Uma vez criada uma variável do tipo da *struct*, é possível acessar diretamente seus membros.
- Para isso, é só colocar um ponto "." entre o nome da variável e seu membro:

```
apple.weight    banana.weight    melon.weight  
apple.price     banana.price     melon.price
```

# Structs [1]

```
struct manga_t
{
    char title[30];
    int year;
} mine, yours;

void printManga (struct manga_t manga)
{
    printf("\nManga Title:\n%s", manga.title);
    printf("\nManga Year of Release:\n%d\n", manga.year);
}
```

# Structs [1]

```
int main ()
{
    strcpy(mine.title, "One Piece");
    mine.year = 1997;

    scanf("%s", yours.title);
    scanf("%d", &yours.year);

    printManga(mine);
    printManga(yours);
    return 0;
}
```

# Structs [1]

- É possível observar que as variáveis `yours.year` é um inteiro e `yours.name` é uma *string*, ambos totalmente válidos
- Além disso, as variáveis *mine* e *yours* são, por si só, também variáveis (do tipo `manga_t`)
- Por isso, é possível passá-las como argumentos de função, verificar seu tamanho entre outras coisas

## Structs [5]

- É possível inicializar uma struct com o uso de chaves, passando os valores na ordem exata da declaração dos argumentos

```
struct manga_t other = {"Yu Yu Hakusho", 1990};
```

# Structs [1]

→ Como *structs* são tipos de dados, elas também podem ser usadas como tipos de vetores.

```
struct manga_t {  
    char title[30];  
    int year;  
} manga[2];
```

# Structs [1]

- É possível observar que as variáveis `yours.year` é um inteiro e `yours.name` é uma *string*, ambos totalmente válidos
- Além disso, as variáveis *mine* e *yours* são, por si só, também variáveis (do tipo `manga_t`)
- Por isso, é possível passá-las como argumentos de função, verificar seu tamanho entre outras coisas

# Structs [1]

- Também é possível fazer ponteiros de *structs*!
- As operações seguem o mesmo princípio de qualquer outra variável
- Mas existe uma diferença:
  - ◆ Para acessar os valores de uma variável de ponteiro de *struct* usa-se o operador “->” no lugar de “.”
  - ◆ Ou *(\*pstruct).value*
- Os dois trechos de código a seguir são equivalentes:

# Structs [1]

```
struct manga_t amanga;  
struct manga_t *pmanga;  
pmanga = &amanga;  
  
strcpy(pmanga->title, "Bleach");  
pmanga->year = 2001;  
  
printManga(pmanga);
```

```
struct manga_t amanga;  
struct manga_t *pmanga;  
pmanga = &amanga;  
  
strcpy((*pmanga).title, "Bleach");  
(*pmanga).year = 2001;  
  
printManga(pmanga);
```

# Structs [1]

- O operador de seta “->” **NÃO** é a mesma coisa que
  - ◆ *\*pmanga.title*
    - Este é a mesma coisa que *\*(pmanga.title)*
    - Ambos acessariam o valor apontado pelo ponteiro hipotético *title* da estrutura *pmanga*
      - Mas *title* não é do tipo ponteiro!

# Structs [1]

- É possível fazer estruturas aninhadas!
  - ◆ Ou seja, uma *struct* que tem outra *struct* como variável
- Com isso é possível criar diversos tipos complexos de dados com uma organização de código muito melhor!

# Structs [1]

```
struct manga_t{
    char title[40];
    int year;
} manga;

struct anime_t{
    struct manga_t manga;
    char studio[30];
    int year;
} anime;

void printManga (struct manga_t manga);
void printAnime (struct anime_t anime);
```

# Structs [1]

```
int main () {  
  
    strcpy(manga.title,  
           "Kono Subarashii Sekai ni Shukufuku o!");  
    manga.year = 2012;  
  
    strcpy(anime.studio, "Studio Deen");  
    anime.year = 2016;  
  
    anime.manga = manga;  
    printAnime(anime);  
    return 0;  
}
```

# Structs [1]

```
void printManga (struct manga_t manga){  
    printf("\nManga Title:\n%s\n", manga.title);  
    printf("\nManga Year of Release:\n%d\n", manga.year);  
}  
  
void printAnime (struct anime_t anime){  
    printManga(anime.manga);  
  
    printf("\nAnime Studio:\n%s\n", anime.studio);  
    printf("\nAnime Year of Release:\n%d\n", anime.year);  
}
```

## Structs [2]

- Também é possível realizar alocação dinâmica de variáveis de *structs*
- O processo é exatamente igual com qualquer outra variável
  - ◆ Colocando o tipo como struct
- Veja o trecho de código a seguir:

## Structs [2]

```
struct manga_t *pmanga;  
  
pmanga = (struct manga_t*) malloc(sizeof(struct manga_t)*1);  
  
strcpy(pmanga->title, "Kubera");  
pmanga->year = 2010;  
  
printManga(pmanga);
```

## Structs [3, 4]

- Ok... é bem chato ficar digitando “*struct*” toda hora, né?
- Em C é possível dar um “apelido” (*alias*) para um tipo
  - ◆ Isso faz com que um tipo seja identificado com um nome diferente
- É só usar a palavra-chave *typedef* seguida pelo tipo e seu novo nome
  - ◆ Pode-se usar em tipos primitivos ou *structs*, por exemplo

## Structs [3, 4]

```
typedef char C;  
  
typedef unsigned int WORD;  
  
typedef char * pChar;  
  
typedef char field [50];  
  
typedef struct {  
    char title[30];  
    int year;  
} manga_t;
```

## Structs [3, 4]

```
C mychar, anotherchar, *ptc1;
```

```
WORD myword;
```

```
pChar ptc2;
```

```
field name;
```

```
manga_t manga;
```

FILE

## FILE [6, 7]

- Existe uma *struct* muito importante chamada FILE, que é essencial para a manipulação de arquivos
- Os conteúdos internos dela não são muito relevantes para a maioria dos programadores, mas vamos apresentar a seguir:

# Structs [7]

```
typedef struct
{
    short level;
    short token;
    short bsize;
    char fd;
    unsigned flags;
    unsigned char hold;
    unsigned char *buffer;
    unsigned char *curp;
    unsigned istemp;
}FILE ;
```

<b>Member</b>
<b>Use / Function</b>
<b>level</b>
<b>Fill / Empty level of Buffer</b>
<b>token</b>
<b>Validity Checking</b>
<b>bsize</b>
<b>Buffer size</b>
<b>fd</b>
<b>File descriptor using which file can be identified</b>

<b>flags</b>
<b>File status flag</b>
<b>hold</b>
<b>ungetc character if no buffer space available</b>
<b>buffer</b>
<b>Data transfer buffer</b>
<b>curp</b>
<b>Current active pointer</b>
<b>istemp</b>
<b>Temp. File indicator</b>

Fonte: [7]

## FILE [6]

- Os conteúdos de FILE são feitos para serem acessados somente por funções padrões de C, como as da *stdio.h*
- A alocação de memória dele é feita automaticamente
  - ◆ Através de funções como *fopen()*
- Assim como a liberação de memória após a chamada de *fclose()*
- Além disso, *stdin*, *stdout* e *stderr* são do tipo *FILE*

## FILE [6]

```
int main() {
    FILE *pFile;
    char buffer [100];
    pFile = fopen ("aizen.txt" , "r");
    if (pFile == NULL) perror ("Error opening file");
    else {
        while ( ! feof (pFile) ) {
            if ( fgets (buffer , 100 , pFile) == NULL ) break;
            fputs (buffer , stdout);
        }
        fclose (pFile);
    }
    return 0;
}
```

## FILE [6]

- *fopen(filename, mode)* é responsável por abrir um arquivo "*filename*" e retornar um ponteiro para *FILE*
- ◆ Existem diversos modos de abertura de arquivos, vamos ver mais exemplos aula que vem
- *feof(stream)* é uma função que retorna 0 enquanto o fim do arquivo não é encontrado
- ◆ Este fim é representado pela macro EOF

## FILE [6]

→ *fgets(str, num, stream)*

- ◆ é uma função utilizada para ler um vetor de *chars* "*str*" do arquivo "*stream*", até um número máximo determinado de caracteres "*num*"

→ *fputs(str, stream)*

- ◆ é uma função que escreve uma string *str* para um arquivo *stream*

→ *fclose(stream)* fecha o arquivo *stream* e libera memória

# Referências

1. <http://www.cplusplus.com/doc/tutorial/structures/>
2. [https://www.learn-c.org/en/Dynamic\\_allocation](https://www.learn-c.org/en/Dynamic_allocation)
3. [http://www.cplusplus.com/doc/tutorial/other\\_data\\_types/](http://www.cplusplus.com/doc/tutorial/other_data_types/)
4. <https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/typedef.html>
5. <https://www.geeksforgeeks.org/structures-c/>
6. <http://www.cplusplus.com/reference/cstdio/FILE/>
7. <http://www.c4learn.com/c-programming/c-file-structure-and-file-pointer/>