

Revisão Pontos Importantes:

Structs: É um novo tipo que permite a criação de tipos compostos de tipos já definidos em uma só estrutura local.

Pode ser usado como unidade e é muito útil no uso de dados organizados:

<pre>Struct nome { // tipos já existentes };</pre>	≡	<pre>typedef struct { // tipos já existentes } nome;</pre>
↓		↓

Struct nome nome var;		nome nome var;
----------------------------------	--	----------------

O acesso a um tipo de uma struct usa-se o (.).

Além de poder incluir uma struct, pode-se copiar para outro com apenas (=).

→ União:

Semelhante à struct, mas seus membros não podem coexistir, apenas

um por vez. (Todos os membros compartilham o mesmo espaço de memória)

Sintaxe similar:

```
typedef union {  
    // tipos existentes  
} nome;  
↓
```

nome nome_var;

A union deixa a memória necessária para o maior membro da union. [MEIO INÚTIL!]

→ Enum:

define numeros constantes para um variável, associando a um nome.

Sintaxe: $\begin{matrix} 0 & 1 & \dots \\ \uparrow & \uparrow & \\ \text{valor} & \text{valor} & \dots \end{matrix}$
typedef enum { $\begin{matrix} 0 & 1 & \dots \\ \uparrow & \uparrow & \\ \text{valor} & \text{valor} & \dots \end{matrix}$ } nome;

[MAIS INÚTIL AINDA!]

→ Funções (Passagem por cópia e referência): [IMPORTANTE]

20 Cópia: uma função que recebe valores e faz uma cópia para, uma vez altera o original.

20 Referência: altera o variável original (poupa memória e recursos)

Para usar os valores que o operador (&) retorna o endereço de memória de uma variável.

Para ter variáveis que trabalhem com endereços, usamos ponteiros:

tipo variável * Nome variável;
↓
operador de ponteiro. nome variável = & valor; →

↪ declaração

↪ atribuição

* nome variável.

↪ uso

[Por padrão vetores e matrizes já são ponteiros que apontam para o 1º elemento]

↳ Structs como parâmetros:

Structs no padrão são normais, mas se precisa usar, a referência é feita de maneira similar, apenas o acesso de seus membros muda de operador;

```
Struct  
int x, y;  
} Ponto
```

void incrementa (Ponto *P) {

```
    P->x++;  
    P->y++;  
}
```

main { Ponto p1 = {1, 2}; incrementa (&p1); }

$P \rightarrow X++$ equivalente a $(*P).X++$;

→ Recurção:

↳ recurso

↳ iterativo

A recurso (ou for melhorado) é uma função que se chama diversas vezes até resolver (*) um problema de forma controlada.

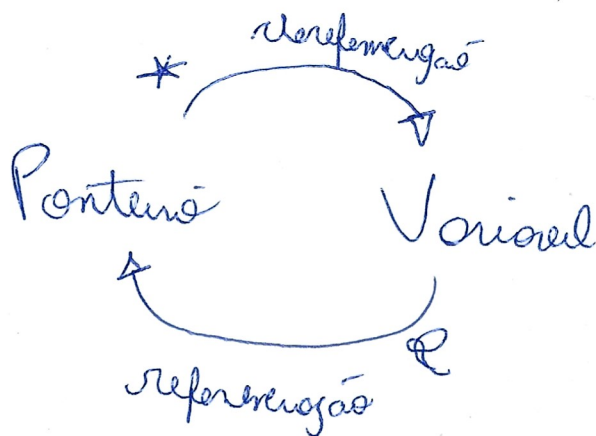
Ela não realiza uma tarefa e se chama até que chegue ao caso base, processo chamado Stack, quando for desimpilhado ele não desdobra os valores e será o destack e retorna a 1ª chamada com a tarefa realizada.

→ Ponteiro:

// " " " " " "

Cada tipo de ponteiro segue seu tipo de variável;

Um ponteiro que aponta para nada é inicializado com NULL;



Um ponteiro só
pode receber dados
(endereços) de uma
variável de mesmo tipo dele

Aritmética de ponteiros:

Sabe ponteiros pode soma e subtrair, mas deve-se observar os tipos de dados trabalhados, afinal isso impacta no peso da variável. (Será possível comparar 2 ponteiros).

Ponteiro genérico:

É aquele que aponta para qualquer tipo, mas deve-se usar o casting para converter no tipo trabalhado.

void* ptr genérico; (vale 1 byte a sua aritmética!)

Vetores:

Já são ponteiros para o 1º endereço de memória do vetor. (Forma referência)

Ponteiros → Ponteiros:

Em C pode-se fazer uma ou mais níveis de ponteiros que apontam para lugares e de jeitos diferentes:

$Ptr \leftarrow *Ptr \leftarrow **Ptr \dots$

GREAT TIPS:

- 1º) Casting é necessário de float para int;
- 2º) vetores do tipo: $\text{char} \times [n][m]$ são possíveis
opções: ~~array~~ $\times [] [m]$ nas funções.
- 3º) vetores do tipo: $\text{char} \times [n]$ são possíveis opções:
 $\times []$ nas funções. (não só restrito ao char!)
- 4º) Sempre que possível use-se os recursos de return, break e
etc... eles eliminam trechos grandes, além disso em casos
dores que se formam alternas faça algo, se não mantenha como
está o fluxo do programa;
- 5º) funções retornam valores reais se forem por referência no
seu contexto via ponteiros;
- 6º) funções podem retornar Structs e etc...
- 7º) Foi o melhor amigo do programador!!!
- 8º) Setbuff é necessário em vários casos, use ele sempre.

ALUNO: Gabriel F.F. de Souza - R.A: 2669480 - Prof. Maurício Galvão