Módulo 8 - Unions, structs, tipos de dados

Union

Em linguagem C, os "unions" (uniões) são estruturas de dados que permitem armazenar diferentes tipos de dados em uma única variável, ocupando o mesmo espaço de memória.

O código abaixo mostra um exemplo onde um membro do tipo `short int` (2 bytes), que chamei de “completo”, foi montado utilizando dois bytes diferentes (valores). Este tipo de lógica é muito comum em sistemas de baixo nível para gerenciar conteúdo de buffers e de memória.

```
#include <stdio.h>

union Dado{
    short int completo;
    char valores[2];
};

int main(){
    union Dado valor;

    valor.valores[0] = 0x64;
    valor.valores[1] = 0x64;

    printf("Valor 0: %x\n", valor.valores[0]);
    printf("Endereco 0: %u\n", &valor.valores[0]);
    printf("Valor 0: %x\n", valor.valores[1]);
    printf("Endereco 0: %u\n", &valor.valores[1]);
```

```
    printf("Valor completo: %x\n", valor.completo);
    printf("Endereco completo: %u\n", &valor.completo);

    return 0;
}
```

Uma aplicação prática com `union` é receber um valor de mais de um byte e organizá-lo em um inteiro na memória. Pela forma abaixo, o algoritmo desloca os bits de high para a parte mais alta e faz um OR bit a bit com a parte mais baixa. Isso consome muito processamento, já que exige vários passos, mas resolve o problema.

```
// Parte alta do inteiro
unsigned char high = 0xAA;

// Parte baixa do inteiro
unsigned char low = 0xBB;

int main(void){
        unsigned short result; // inteiro de 2 bytes

        // Transforma os dois char em 1 int
        // Deste modo requer vários passos da CPU
        result = ((high<<8) | low);
        printf("%X", result);
}
```

Na forma abaixo, o processamento necessário é praticamente zero:

```
union{
        unsigned char buf[2];
        unsigned int result;  // CONSIDERANDO INT DE 2 BYTES
} data;

unsigned char high = 0xAA;
unsigned char low = 0xBB;
```

```
int main(void){
        data.buf[0] = low;
        data.buf[1] = high;

        printf("%X", data.result);
        return 0;
}
```

Enum

`Enum` transforma um conjunto de identificadores (ou palavras-chave) definidas pelo usuário em uma sequência de números inteiros

```
#include <stdio.h>

enum sequencia {zero, um, dois, tres, quatro};

int main(){

    printf("%d\n", zero);
    printf("%d\n", um);
    printf("%d\n", dois);
    printf("%d\n", tres);
    printf("%d\n", quatro);

    return 0;
}
```

O identificador que você define passa a funcionar como um inteiro, ou seja, ele age como um inteiro em qualquer operação

```
printf("%d", um + dois);
```

As formas acima, no caso, são a padrão da linguagem. Mas o programador pode definir um ponto inicial da sequência (somente o ponto inicial, não o final), a `enum` vai seguir incrementando os valores normalmente caso existam mais identificadores a frente.

```
#include <stdio.h>

enum sequencia {zero, um, dois = 20, tres, quatro};

int main(){

    printf("%d\n", zero); // 0
    printf("%d\n", um); // 1
    printf("%d\n", dois); // 20
    printf("%d\n", tres); // 21
    printf("%d\n", quatro); // 22

    return 0;
}
```

Você pode definir uma variável do tipo desta estrutura, mas ela não terá nada de especial, será somente uma variável comum do tipo int.

```
enum sequencia valor;

printf("Valor: %d\n", valor); // Algum valor genérico
printf("Size valor: %d\n", sizeof(valor)); // 4 bytes
```

Struct

É um tipo de variável de estrutura. Quando declarada, o compilador aloca memoria para acomodar cada um de seus dados. As variáveis que formam a estrutura são chamadas membros da estrutura.

Uma estrutura pode ser muito parecida com uma `union` em um primeiro momento. Mas elas são completamente diferentes. Uma `union` faz com que seus dados ocupem o

mesmo endereço de memória, enquanto uma `struct` aloca memória para cada um de seus dados.

```
// Estrutura nomeada two_int
struct two_int {
        int a;
        int b;
};

// Instanciação da estrutura (alocação em memória)
struct two_int x;

// Acesso de seus membros
x.a = 10;
```

Perceba que os endereços de memória de a e de b passam a ser distintos. Portando, a `struct` ocupará em memória o equivalente a soma de todas suas variáveis. Se você declara uma `struct` de 4 inteiros de 4 bytes, então a `struct` ocupará 16 bytes sempre que for criada.

Sendo variáveis como qualquer outra. É possível atribuir o valor de uma `struct` à outra:

```
struct {
        int a;
        int b;
} x, y;

x.a = 10;
y = x;
printf("%d", y.a); // 10
```

A atribuição de valores inteiros é algo bem intuitivo, porém, não é possível passar strings da forma como estamos acostumados. Isso porque um array dentro de uma estrutura só recebe um caracter por vez. Então as únicas formas de popular ele com uma string seria com um laço de repetição ou com uma função strcpy().

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

// Caso vc queira varias structs ao longo do código
struct addr {
    char nome[30];
    char rua[30];
    int numero;
    char cidade[20];
    char estado[30];
    char cep;
};
// Você vai declarando elas ao longo do código
struct addr addr_info;

void main(void){
    strcpy(addr_info.nome, "Victor");
    strcpy(addr_info.estado, "Paraná");

    printf("%s reside no estado do %s\n", addr_info.nome, addr_
}
```

Até o momento que o `struct` foi criado, não existe alocação de memória acontecendo. Ela só acontece quando é definido uma variável com este `struct`.

Assim como qualquer tipo de dado, é possível definir um array do tipo de sua `struct`, veja o exemplo do código a seguir:

```
struct {
        int a;
        int b;
} x[10]

x[0].a = 5;
```

```
x[1].b = 10;
x[2].c = 15;
```

Passando struct como parâmetro de função

Há várias formas de se passar uma estrutura para uma função. Uma delas é passando o elemento da estrutura diretamente, desta forma, o comportamento é igual a o de uma variável comum.

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

struct person{
    char nome[30];
    char idade;
} p1;

void fala_oi(char *nome, char idade){
        printf("Oi, %s, voce tem %d anos", nome, idade);
}

int main(){
    strcpy(p1.nome, "Victor");
    p1.idade = 20;

    fala_oi(p1.nome, p1.idade);

    return 0;
}
```

E assim como em variáveis, também é possível passar diretamente o endereço do elemento da estrutura (passagem por referência), para modificar diretamente o seu valor.

Passando estruturas inteiras para funções

Há também a possibilidade de passar uma estrutura inteira para a função. Você precisará declarar o argumento da função como sendo a `struct` que ela irá receber.

```
void fala_oi(struct person pessoa){
    printf("Oi, %s, voce tem %d anos", pessoa.nome, pessoa.idade
}

fala_oi(p1);
```

No caso acima, a declaração da estrutura teria que ser global. Se a mesma tivesse sido declarada em main, então ela não estaria visível para `fala_oi()` utilizar sua estrutura. Geralmente, estruturas são declaradas em escopos globais, então não haveria problema em relação a isso.

Passar a `struct` por valor para uma função pode não ser uma boa prática. Isso porque leva tempo para a nova estrutura (a que será utilizado pela função) ser realocada na memória. Para resolver este problema é possível passar o endereço da estrutura. Antes disso, conheceremos um novo operador:

```
struct person{
    char nome[30];
    char idade;
} p1;

struct person *ponteiro_struct;

ponteiro_struct = &p1;

// Operador seta
printf("%s %d anos", ponteiro_struct->nome, ponteiro_struct->ida
```

Passando dentro de uma função

```
void fala_oi(struct person *pessoa){
    printf("Oi, %s, voce tem %d anos", pessoa->nome, pessoa->ida
```

```
}
fala_oi(&p1);
```

Typedef

`Typedef` lhe permite criar um novo nome a um novo tipo de dado definido pelo programador. Vimos que, sempre que necessário criar uma estrutura, você deverá passar o nome completo dela, tornando o processo trabalhoso as vezes. Primeiro veja o exemplo de transformação de um tipo int para um tipo inteiro

```
typedef int inteiro;

inteiro valor;

printf("%d", sizeof(valor));
```

Da para fazer isso também com `unions`, `structs` e `enums`.

```
typedef union Dado {
    short int completo;
    char valores[2];
} meu_dado;

meu_dado valor; // meu_dado se torna um tipo de dado para o comp

typedef struct mystruct {
        unsigned x;
        float f;
} mystruct;

mystruct s;
```

Extern

A palavra-chave `extern` em linguagem C é usada para declarar uma variável ou função que foi definida em outro local (em outro arquivo-fonte ou em outra unidade de tradução) e torná-la visível no arquivo-fonte atual. Ela informa ao compilador que a definição da variável ou função está em algum outro lugar, permitindo que você use esses símbolos em seu código sem a necessidade de fornecer a definição real.

A utilização do `extern` geralmente ocorre em duas situações principais:

1. **Variáveis globais**: Quando você deseja acessar uma variável global que foi definida em outro arquivo-fonte. Por exemplo:

```
// No arquivo source1.c
int globalVariable = 42;

// No arquivo source2.c
extern int globalVariable; // Declaração externa
```

Neste exemplo, `extern` é usado no arquivo `source2.c` para informar ao compilador que a variável `globalVariable` é definida em outro lugar (no arquivo `source1.c`). Isso permite que você acesse a variável `globalVariable` em `source2.c`.

2. **Funções**: Quando você deseja chamar uma função que está definida em outro arquivo-fonte. Por exemplo:

```
// No arquivo source1.c
void myFunction() {
    // Implementação da função
}

// No arquivo source2.c
extern void myFunction(); // Declaração externa

int main() {
    myFunction(); // Chamada da função
    return 0;
}
```

Neste caso, `extern` é usado para declarar a função `myFunction` no arquivo `source2.c`, permitindo que você a chame no `main`.

É importante notar que `extern` é frequentemente usado em arquivos de cabeçalho (header files) para declarar variáveis ou funções que serão compartilhadas entre vários arquivos-fonte de um projeto. Isso ajuda a manter a consistência e evitar erros de ligação (linker) durante a compilação.

Além disso, ao usar `extern`, o linker é responsável por encontrar a definição real do símbolo (variável ou função) durante a fase de ligação, portanto, é importante garantir que a definição exista em algum lugar no projeto.

Static

Local Static

Uma variável do tipo estática é alocada de uma forma diferente em memória. Geralmente, ela não fica na mesma região de memória que as outras variáveis de seu programa, além de ser alocada antes mesmo da chamada da main.

Por este motivo, mesmo que dentro de um escopo específico (como a função incrementa abaixo), ela preservará seu valor sempre que for acessada. Ela estará sempre alocada em uma região especial de memória.

```
void incrementa(){
    static int var = 0;
    var++;
    printf("%d\n", var);
}

int main(){
    incrementa();
    incrementa();
    return 0;
}
```

Variáveis estáticas são inicializadas em 0, diferente de uma variável comum

```
static int x;
int y;
printf("x- %d y- %d", x, y);
```

Só podem ser inicializadas com constantes literais. Uma função, mesmo que retorne um valor constante, não é considerada uma constante literal, algo que pode resultar em um erro. Isso ocorre porque a variável `static` será alocada antes mesmo da execução de main.

```
#include <stdio.h>

int inicializa(){
    return 50;
}

void main(){
    static int i = inicializa();
    printf("%d", i);
}
```

Variáveis estáticas não devem ser declaradas em estruturas. Em uma estrutura, todas as variáveis são alocadas em conjunto, então declarar uma `static` dentro da estrutura faria com que esta variável fosse alocada em outro local, gerando um erro. Mas a estrutura inteira pode ser declarada como `static`.

```
struct {
    static int x;
    float y;
} my_struct;
```

Global static

Uma função é naturalmente de escopo global em C. Então, em projetos de várias folhas, ela ficaria visível em todas as folhas. Declarar uma função como `static` faz com que ela seja visível somente para o escopo da folha onde está declarada.

```
static void fazAlgo();
```

Se a função for declarada como `static`, ela só poderá ser chamada dentro da folha onde foi declarada. Qualquer projeto que inclua esta folha não poderá chamar esta função, pois ela não existirá fora da folha

Volatile

O modificador `volatile` em linguagem C é utilizado para indicar ao compilador que uma variável pode ser alterada a qualquer momento por meios externos ao código, como interrupções de hardware ou outros threads. Isso impede o compilador de realizar otimizações que poderiam causar resultados inesperados quando se trabalha com variáveis que podem ser modificadas fora do controle direto do código.

Quando uma variável é declarada como `volatile`, o compilador deve garantir que todas as leituras e gravações dessa variável são realizadas diretamente na memória, sem realizar otimizações que possam causar problemas de concorrência.

Aqui está um exemplo de declaração de uma variável volátil:

```
volatile int minhaVariavel;
```

Um caso comum de uso do modificador `volatile` é quando se trabalha com registradores de hardware ou variáveis que podem ser alteradas por interrupções. Por exemplo, em programação de sistemas embarcados:

```
volatile int *enderecoRegistrador = (int *)0x12345678;
```

Ao usar o modificador `volatile`, você informa ao compilador que deve tratar esse ponteiro como uma referência a um registrador volátil, evitando otimizações que poderiam causar problemas de sincronização.

Const

O modificador `const` em linguagem C é utilizado para indicar que uma variável é uma constante, ou seja, seu valor não pode ser alterado após a inicialização. Quando você

declara uma variável como `const`, está indicando ao compilador que qualquer tentativa de modificação do valor da variável resultará em um erro de compilação.

Aqui está um exemplo de como usar o modificador `const`:

```
const int minhaConstante = 42;
```

Neste exemplo, `minhaConstante` é uma variável inteira que foi declarada como constante e inicializada com o valor 42. Qualquer tentativa de alterar o valor de `minhaConstante` resultará em um erro de compilação.

O uso de `const` não apenas fornece uma indicação clara de que a variável não deve ser modificada, mas também permite que o compilador realize otimizações adicionais, pois ele pode assumir que o valor da constante não mudará durante a execução do programa.

Além de ser usado com tipos de dados simples, o modificador `const` também pode ser aplicado a ponteiros. Existem duas formas de usar `const` com ponteiros:

1. **Ponteiro para dados constantes:**

```
const int *ptr;
```

Isso indica que o valor apontado por `ptr` é constante, ou seja, você não pode modificar o valor através do ponteiro.

2. **Ponteiro constante para dados:**

```
int const *ptr;
```

Isso indica que o próprio ponteiro `ptr` é constante, ou seja, uma vez que ele aponta para um endereço de memória, esse endereço não pode ser alterado.

Lembre-se de que usar `const` não torna a variável imutável na memória; ela apenas impede a modificação através da variável direta ou do ponteiro constante. Se a constante for um objeto composto (como uma estrutura ou um array), a imutabilidade se aplica ao objeto como um todo, e não necessariamente a cada elemento individual.

Register

Em linguagem C, a palavra-chave `register` é um modificador de armazenamento que pode ser usada ao declarar uma variável. No entanto, é importante notar que o modificador `register` é, na prática, menos utilizado nas versões mais recentes do padrão C (a partir do C99) e pode até mesmo ser ignorado pelo compilador, que possui otimizações automáticas.

A intenção original do modificador `register` era sugerir ao compilador que armazenasse a variável em um registrador de CPU para otimizar o acesso rápido durante a execução do programa. No entanto, os compiladores modernos são geralmente capazes de otimizar automaticamente o uso de registradores, tornando a diretiva `register` menos relevante.