Laboratório de programação

Tipos definidos pelo programador

Universidade Estadual Vale do Acaraú – UVA

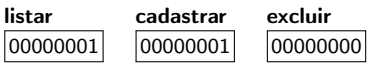
Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

- Bitfields
- Unions
- Enums
- Typedef

- A linguagem C permite que você controle os dados ao nível de bits.
- Digamos que você precise de um `struct` para conter vários valores sim/não. Você poderia criar um `struct` com uma série de `ints` ou `chars`:

```
struct Permicoes {
    unsigned char listar;
    unsigned char cadastrar;
    unsigned char excluir;
};
```



- Isso funciona, mas os campos `char` irão ocupar muito mais espaço do que o *único bit* que você precisa para valores **true**/**false**.

- Os `bitfields` foram criados para que você possa definir quantos bits um campo individual vai armazenar.
- Digamos que você precise de um `struct` para conter vários valores sim/não. Você poderia criar um `struct` com uma série de `ints` ou `chars`:

```
struct Permicoes {
    unsigned char listar:1;
    unsigned char cadastrar:1;
    unsigned char excluir:1;
};
```

00000<**listar**,**cadastrar**,**excluir**>

0000110

- Se houver uma sequência de `bitfields` o computador pode **comprimi-los** para economizar espaço. Então se houver oito `bitfields` de um único bit, o computador pode armazená-los em um único byte.

De quantos bits eu preciso?

- Bitfields podem ser usados para armazenar uma sequência de valores verdadeiro/falso, mas também são úteis para outros valores de opções limitadas.
- Se quiser armazenar um número do mês do ano em um `struct`, você sabe que ele terá um valor de 1 até 12.
- Esses valores podem ser armazenados em apenas **4 bits**.

```
unsigned char numeroDoMes:4;
```

- Bitfields podem ser usados para armazenar uma sequência de valores verdadeiro/falso, mas também são úteis para outros valores de opções limitadas.
- Se quiser armazenar um número do mês do ano em um `struct`, você sabe que ele terá um valor de 1 até 12.
- Esses valores podem ser armazenados em apenas **4 bits**.

```
unsigned char numeroDoMes:4;
```

Número de bits	Faixa de valores
1	0,1
2	0, . . . , 3
3	0, . . . , 7
4	0, . . . , 15
n	$0, \ \ldots, \ 2^n - 1$

```c
struct Permicoes {
    unsigned int listarProduto:1;
    unsigned int cadastrarProduto:1;
    unsigned int excluirProduto:1;
    unsigned int :5;
    unsigned int listarUsuario:1;
    unsigned int cadastrarUsuario:1;
    unsigned int excluirUsuario:1;
} permicoes;
```

- Os campos de bits **podem** ser criados **sem nome**.
  - São utilizados para preencher uma estrutura de modo a adequá-la a um layout específico.
  - Não podem ser acessados nem inicializados.
  - O comprimento do campo de bits **não deve** exceder o número **total de bits** do tipo da variável utilizada na declaração.

```
1 struct Permicoes {
2     unsigned int listarProduto:1;
3     unsigned int cadastrarProduto:1;
4     unsigned int excluirProduto:1;
5     unsigned int :0;
6     unsigned int listarUsuario:1;
7     unsigned int cadastrarUsuario:1;
8     unsigned int excluirUsuario:1;
9 } permicoes;
```

- **Devem** ser declarados como `unsigned int`. **Podem** ser usados os modificadores `signed` e `unsigned`. Se o campo for do tipo `signed int` seu comprimento deverá ser maior do que 1 (um bit será usado para o sinal).
- Campos de bits **podem** ter tamanho zero
  - Não podem possuir nome
  - Indica que nenhum campo de bits adicional deve ser colocado dentro no mesmo `int`.

Exercício 27

Crie um programa para coletar informações sobre as visitas ao aquário da cidade.

a) Crie um vetor com o 50 posições para guardar as respostas dos visitantes, abaixo segue a estrutura que deve ser utilizada, você deve completá-la com os números de bits adequados:

```
struct Formulario {
    unsigned int primeiraVisita:(número de bits);   //sim ou nao
    unsigned int visitarNovamente:(número de bits);//sim ou nao
    unsigned int dedosPerdidos:(número de bits);    //quantidade
    unsigned int criancaPerdida:(número de bits);   //sim ou nao
    unsigned int quantosDias:(número de bits);      //duracao
};
```

b) Pergunte quantas pessoas irão responder à pesquisa (máximo 50) e inicie a coleta das informações.

Continuação...

c) O sistema deverá apresentar um questionário com as seguintes perguntas:
   1. É sua primeira visita?
   2. Pretende nos visitar novamente?
   3. Número de dedos perdidos no aquário das piranhas:
   4. Você perdeu uma criança na exibição dos tubarões?
   5. Quantos dias da semana você passaria aqui se fosse possível?

d) Crie uma função que receba um vetor da estrutura `Formulario` e imprima um resumo das informações: **(1)** quantas pessoas visitaram pela primeira vez, **(2)** quantas pretendem visitar novamente, **(3)** número total de dedos perdidos no aquário das piranhas e a **(4)** média de dias que as pessoas passariam no aquário.

- Às vezes, o mesmo tipo de coisa precisa de tipos diferentes de dados.
  1. *6 maçãs* → quantidade de maçãs
  2. *1,5kg de carne* → quantidade de carne
  3. *0,5l de leite* → quantidade de leite
- Digamos que queira registrar uma *quantidade* de algo, e esta quantidade pode ser uma **contagem**, um **peso** ou um **volume**.

```
1 union Quantidade {
2     short unidades;
3     float peso;
4     float volume;
5 };
```

- Isso não é uma boa ideia por diversos motivos:
  - Vai ocupar mais espaço na memória.
  - Talvez alguém atribua mais que um valor.
  - Não há nada que se chama "quantidade".
- Seria muito útil se fosse possível especificar algo chamado "quantidade" em um tipo de dados e, depois, decidir para cada dado individual se vai registrar uma contagem, um peso ou volume.

- Cada vez que você cria uma instância de um struct, o computador coloca os campos na memória em sequência:

char *nome	int idade	float imc

- Um **union** é diferente, o union vai usar o espaço de apenas um dos campos na sua definição.
- O computador só dá espaço suficiente ao union para o maior dos campos.

Quantidade (pode ser um short ou um float)

```c
union Quantidade {
    short unidades;
    float peso;
    float volume;
};
```

- Quando se declara uma variável union, há algumas maneiras de atribuir um valor a ela.
  1. **Estilo C89 para o primeiro atributo**: se o union vai armazenar um valor para o primeiro campo, você pode usar a notação C89:

```
1 union Quantidade c = {4};
```

  2. **Inicializadores designados**: um inicializador designado atribui um valor a um campo pelo nome:

```
1 union Quantidade q = {.peso = 1.5};
```

  3. **Notação de ponto**: a terceira maneira é criar a variável em uma linha e atribuir um valor de campo em outra linha:

```
1 union Quantidade q;
2 q.volume = 3.7;
```

- Unions fequentemente são usados com structs

```
1 struct Fruta {
2     char *nome;
3     char *pais;
4     union Quantidade qtd;
5 };
```

- Pode-se acessar os valores na combinação struct/union usando a notação ponto ou ->:

```
1 struct Fruta uvas = {"uva", "Itália", .qtd.peso = 4.2};
2 printf("Pedido: %2.2fkg de %s\n", uvas.qtd.peso, uva.nome);
```

Warning!

O compilador não vai poder manter o controle dos campos que são atribuídos e lidos em um union, então nada impede que você atribua um valor a um campo e leia de outro, o que as vezes pode ser um **grande problema**.

```c
#include <stdio.h>
union Cupcake {
    float peso;
    int qtd;
};

int main() {
    union Cupcake pedido = {2};
    printf("Quantidade de cupcakes: %d\n", pedido.qtd);
    return 0;
}
```

Exercício 28

Defina uma estrutura da seguinte forma

```
union Generico {
    int vInt;
    char vChar;
    float vFloat;
};
```

Crie a função `void printArray(union Generico g[], int tamanho, int tipo)` que recebe um vetor do tipo `struct Generico`, o tamanho do vetor e o `tipo` (valores 1,2 ou 3 para int, char ou float respectivamente). A função deve imprimir o vetor de acordo o tipo selecionado;

Exercício 29

Observe o programa `union_encript.c` a seguir, ele usa uma união para armazenar conteúdo em forma de texto e depois apresentá-lo de forma numérica. O programa usa isto para codificar uma mensagem de 20 caracteres em 5 campos numéricos. Crie um programa `union_decript.c` que realiza o processo inverso, leia 5 números inteiros (mensagem codificada) e apresente a mensagem decodificada.

Exemplo:

1936614765 1835362145 1667592992 1635018098 0 $\rightarrow$ "mensagem secreta"

Código 1: union_encript.c

```c
#include <stdio.h>
union Mensagem {
    int campo[5];    // 20 bytes = 5 * 4bytes
    char texto[20]; // 20 bytes = 20 *1byte
};
int main() {
    int i;
    union Mensagem m = {0};
    printf("|-----Mensagem-----|\n ");
    scanf("%[^\n]", m.texto);
    printf("Mensagem codificada\n");
    for (i=0; i<5; i++)
        printf("%d ", m.campo[i]);
    printf("\n");
    return 0;
}
```

- Às vezes, você não quer armazenar um número ou trecho de código, mas sim uma lista de **símbolos**. Por exemplo os dias da semana: SEG, TER, QUA, ...
- O enum foi criado para permitir guardar uma lista de símbolos.

```
1 enum Cor {VERMELHO, VERDE, ROSA};
```

- Qualquer variável definida com um tipo `enum cor` apenas poderá receber uma das palavras-chave da lista.
- Uma variável `enum cor` poderia ser definida assim:

```
1 enum Cor favorita = ROSA;
```

O computador irá atribuir um número para cada símbolo da lista e o enum vai armazenar um número, mas seu código só precisa referenciar os símbolos.

- Podemos usar os enums para manter o controle dos unions da seguinte maneira.

```c
#include <stdio.h>

enum UnidadeDeMedida {UNIDADE, QUILO, LITRO};

union Quantidade {
    short unidades;
    float peso;
    float volume;
};

struct Pedido {
    char *nome;
    char *pais;
    union Quantidade qtd;
    enum UnidadeDeMedida uni;
};
```

```c
void printPedido(struct Pedido fruta) {
    printf("Pedido de Fruta: ");
    if (fruta.uni == LITRO)
        printf("%2.2f litros de %s\n", fruta.qtd.volume, fruta.nome);
    if (fruta.uni == QUILO)
        printf("%2.2f quilos de %s\n", fruta.qtd.peso, fruta.nome);
    if (fruta.uni == UNIDADE)
        printf("%i %s(s)\n", fruta.qtd.unidades, fruta.nome);
}

int main() {
    struct Pedido maca = {"Maçã", "Inglaterra", .qtd.unidades=144,
        UNIDADE};
    struct Pedido uva = {"Uva", "Itália", .qtd.peso=17.6, QUILO};
    struct Pedido suco = {"Suco de laranja", "U.S.A.", .qtd.volume
        =10.5, LITRO};
    printPedido(maca);
    printPedido(uva);
    printPedido(suco);
    return 0;
}
```

- Na definição da enumeração, pode-se definir qual valor aquela constante possuirá.

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

enum Semana {Domingo = 1, Segunda, Terca, Quarta=7, Quinta, Sexta,
     Sabado};

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

- A linguagem C permite que o programador defina os seus próprios tipos com base em outros tipos de dados existentes.
- Para isso, utiliza-se o comando **typedef**, cuja forma geral é:

```
1 typedef tipo_existente novo_nome;
```

- **tipo_existente** é um tipo básico ou definido pelo programador (por exemplo, uma `struct`)
- **novo_nome** é o nome para o novo tipo que está sendo definindo.

- No comando **typedef**, o sinônimo e o tipo existente são equivalentes.

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

typedef int Inteiro;

int main(){
    int x = 10;
    Inteiro y = 20;

    y = y + x;
    printf("Soma = %d\n", y);

    return 0;
}
```

- Imagine a seguinte declaração de uma **struct**:

```
1 struct Usuario {
2     char login[30];
3     int senha;
4 };
```

- A declaração de uma variável c dessa estrutura seria:

```
1 struct Usuario u;
```

- O comando typedef pode ser usado para eliminar a necessidade da palavra-chave **struct** na declaração de variáveis.

```
1 typedef struct Usuario Usuario;
2 //...
3 Usuario u;
```

- O comando **typedef** pode ser combinado com a declaração de um tipo definido pelo programador em uma única instrução.

```
1 typedef struct Usuario {
2     char login[30];
3     int senha;
4 } Usuario;
```

- Como está sendo associado um novo nome à **struct**, seu nome original pode ser omitido da declaração.

```
1 typedef struct {
2     char login[30];
3     int senha;
4 } Usuario;
```

Exercício 30

Utilize as informações a seguir para criar um controle automatizado de uma clínica médica. Sabe-se que essa clínica deseja ter um controle semanal (de segunda a sexta-feira) das consultas realizadas. A cada dia, cada médico pode realizar, no máximo, duas consultas. Considere que serão cadastrados três médicos e cinco pacientes.

```
Paciente(codPac, nome, Endereco, fone)
Medico(codMed, nome, fone, Endereco)
Consulta(numConsulta, dia semana, hora, codMed, codPac)
```

O programa deverá realizar as seguintes operações:

- cadastrar os pacientes, não permitindo dois pacientes com o mesmo código;
- cadastrar os médicos, não permitindo dois médicos com o mesmo código;
- cadastrar as consultas, obedecendo às especificações apresentadas;
- pesquisar as consultas de determinado médico em certo dia da semana (segunda a sexta-feira);
- mostrar um relatório contendo todas as consultas realizadas em um dia.