Estrutura de Programação em C.

Número ISBN: 978-85-917609-2-3

Agradecemos a Deus, aos mestres que fizeram parte de nossa formação ao nossos pais, especialmente ao pai e amigo Pedro Marconi que não está mais entre nós, aos nossos amigos as nossas companheiras e a todos que colaboraram diretamente e indiretamente para conclusão desta obra .

Coleção “ Programação C ”.

Guilherme Marconi

Ricardo Akira Harada

Prefixo Editorial: 917609

ISBN do Anais do ENEAC 2013: 978-85-7426-129-4

1.1 Símbolos usados com ponteiros

Para usar ponteiros, são utilizados os operadores * e &. Estes símbolos possuem diferentes significados na linguagem C, dependendo do contexto. Portanto, um mesmo símbolo pode representar diferentes conceitos. O símbolo * em uma expressão aritmética representa multiplicação. Este símbolo, quando usado na declaração de variáveis, especifica uma variável do tipo ponteiro. Já em comandos de atribuição, este símbolo antecedendo uma variável refere-se a um ponteiro, o qual indica o conteúdo do endereço apontado. De forma similar, o símbolo & pode representar o operador bit-a-bit ou pode representar o endereço de uma variável.

1.2 Aritmética de ponteiros

Ponteiros armazenam endereços de memória. Com ponteiros, pode-se usar os operadores aritméticos de adição e de subtração. Estes operadores aritméticos modificam o conteúdo de um ponteiro, aumentando ou diminuindo o valor do endereço armazenado. O trecho de código a seguir exemplifica o uso de operadores aritméticos com ponteiros. O ponteiro pont_idade aponta para uma variável do tipo int. Supondo que o tipo int ocupe 2 bytes, cada vez que pont_idade é incrementado, seu endereço é acrescido do tamanho de 2 bytes. Este é um aspecto crucial na aritmética de ponteiros. O incremento e o decremento do conteúdo de ponteiros é feito de acordo com o tamanho do tipo base apontado pelo ponteiro. Desta forma, deve-se sempre indicar o tipo apontado por um ponteiro na sua declaração. Em ocasiões especiais, pode-se declarar um ponteiro do tipo void (i.e., declarado como void *pont_void). Este tipo de ponteiro pode ser usado para apontar para diferentes tipos de dados em um mesmo programa. Porém, o controle do aumento e do decremento de um ponteiro void deve ser realizado sem usar aritmética de ponteiros.

```
...
int *pont_idade; // ponteiro do tipo int
...
pont_idade = &idade; // assuma que endereço de idade é 704
pont_idade++; // conteúdo de pont_idade recebe 706
pont_idade+=3; // conteúdo de pont_idade recebe 712
...
```

1.3 Declaração de ponteiros com inicialização

Um ponteiro pode ser inicializado juntamente com a sua declaração. O trecho abaixo inicializa o ponteiro pont_idade com o endereço de memória 704. Note que, em uma declaração de variáveis, *pont_idade não se refere ao conteúdo do endereço apontado. Em contrapartida, em uma linha de comando, tal como em idade = *pont_idade, *pont_idade refere-se ao conteúdo do endereço apontado.

```
...
int *pont_idade = 704; // declaração e inicialização
...
idade = *pont_idade;
...
```

Atribuição de ponteiros

Um ponteiro pode receber o valor de outro ponteiro de forma similar ao que ocorre em um comando de atribuição com outros tipos de variáveis. Isto significa que um ponteiro receberá o endereço de memória armazenado em outro ponteiro. O trecho a seguir ilustra um exemplo de atribuição. Como resultado final, pont_idade2 irá armazenar o endereço 704. Portanto, após o comando de atribuição, ambos os ponteiros apontam para o mesmo endereço.

```
...
int *pont_idade1 = 704;
int *pont_idade2;
...
pont_idade2 = pont_idade1;
...
```

Comparação de ponteiros

De forma similar à atribuição de ponteiros, ponteiros podem ser comparados entre si usando operadores relacionais e de igualdade. Neste caso, compara-se os endereços armazenados nos ponteiros. O trecho a seguir ilustra a comparação de dois ponteiros usando o operador relacional > para decidir a execução de um trecho de programa.

```
...
if (pont_idade1 > pont_idade2)
{
// seqüência de comandos
}
...
```

Ponteiro nulo

Quando um ponteiro não aponta para nenhum endereço válido, o ponteiro é inicializado com o endereço 0. Esta é uma convenção usada por programadores C. Para facilitar o seu uso, a linguagem C define a constante NULL para representar o endereço 0. O trecho de programa a seguir ilustra a inicialização do ponteiro pont_idade com o valor NULL. Sempre que um ponteiro ficar temporariamente sem uso, deve-se atribuir NULL a ele. Caso contrário, erros podem acontecer durante a execução do programa, os quais podem inclusive travar o computador. Por exemplo, um ponteiro que armazena um endereço arbitrário pode estar apontando para uma área de memória que contém o próprio conjunto de instruções do programa. A modificação do conteúdo apontado por este ponteiro irá causar a perda da instrução e, por conseguinte, a execução incorreta do programa ou o travamento do programa. Previne-se este erro atribuindo o valor NULL ao ponteiro sem uso.

```
...
int *pont_idade;
...
pont_idade = NULL; // pont_idade = 0
```

```
...
```

Alocação dinâmica de memória

Quando uma variável de qualquer tipo é declarada, uma porção de espaço da memória é reservada para armazenar o conteúdo desta variável. Para a grande maioria das situações, sabe-se na escrita do programa o tamanho necessário de memória que deve ser alocado para os dados. No entanto, em algumas situações, desconhece- se a quantidade de memória que os dados irão ocupar. Assim, a declaração de variáveis conforme já estudamos não se aplica. Nestas situações, deve-se alocar memória em tempo de execução do programa. A linguagem C oferece funções específicas para o gerenciamento de memória alocada dinamicamente. O uso destas funções é realizado por meio de ponteiros. A função malloc( ) aloca uma porção de memória de tamanho n, onde n é passado como parâmetro. Esta função retorna o endereço da porção de memória alocada, a qual deve ser armazenada em um ponteiro. De forma oposta, a função free( ) libera uma porção de memória alocada dinamicamente, ou seja, esta porção pode ser novamente usada em alocações subseqüentes. Para tanto, deve-se passar como parâmetro da função free( ) o endereço inicial da porção de memória alocada. Este endereço é freqüentemente armazenado em um ponteiro. O trecho de programa abaixo ilustra o uso das funções malloc( ) e free( ). Neste trecho, pont_idade é um ponteiro do tipo int. Portanto, na função malloc( ), solicita-se a alocação de uma porção de memória do tamanho de um tipo int. Este tamanho é determinado pela função sizeof( ). O teste condicional verifica se a alocação de memória ocorreu com sucesso, ou seja, se pont_idade recebeu um valor diferente de 0 (i.e., NULL). O uso da função free() é ilustrado no final do trecho do programa.

```
...
int *pont_idade;
...
pont_idade = malloc(sizeof(int));
if (!pont_idade) // pont_idade == NULL
{
printf("Erro na alocacao de memoria\n");
exit(1);
}
else // pont_idade != NULL
{
printf("O valor de pont_idade e: %p\n", pont_idade);
}
...
// libera a memória quando pont_idade não for mais usado no programa
free(pont_idade);
...
```

Erros comuns com ponteiros

São dois os erros mais comuns que ocorrem no uso de ponteiros. O primeiro deles refere-se a não inicialização de um ponteiro antes de seu primeiro uso. No trecho abaixo, o ponteiro pont_idade não foi inicializado e, portanto, pode conter qualquer endereço de memória. Este

endereço provavelmente será um endereço inválido. O comando de atribuição poderá causar um erro de execução do programa.

```
...
int idade;
int *pont_idade;
idade = *pont_idade;
...
```

O segundo erro refere-se ao esquecimento do uso do operador &. Assim, em um comando de atribuição, a ausência deste operador faz com que a variável ponteiro receba o valor armazenado na variável do lado direito do comando de atribuição, e não o endereço desta variável. No trecho de programa a seguir, o comando correto seria pont_idade = &idade.

```
...
int idade = 10;
int *pont_idade;
pont_idade = idade; // pont_idade recebe 10
...
```

Declaração conjunta de estruturas e variáveis

Usualmente, a declaração de uma estrutura e das variáveis do tipo desta estrutura é realizada separadamente. Por exemplo, no Programa 3, enquanto a declaração da estrutura funcionario é realizada nas linhas 10 a 16, a declaração das variáveis funcionario1 e funcionario2 deste tipo de estrutura é feita nas linhas 19 e 20. Entretanto, é possível efetuar a declaração conjunta de uma estrutura e de suas variáveis. No primeiro trecho de programa a seguir, as variáveis funcionario1 e funcionario2 são declaradas conjuntamente com a estrutura funcionario. Já no segundo trecho, o nome da estrutura é omitido, desde que somente nesta parte do programa esta estrutura é usada.

```
...
struct funcionario
{
int codFunc; // código do funcionário
int idadeFunc; // idade do funcionário
char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
float salFunc; // salário do funcionário
} funcionario1, funcionario2;
...
```

```
...
struct
{
int codFunc; // código do funcionário
int idadeFunc; // idade do funcionário
char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
float salFunc; // salário do funcionário
} funcionario1, funcionario2;
...
```

Uso de typedef para simplificar a declaração de variáveis do tipo estrutura

A palavra reservada typedef permite a criação de novos tipos de dados. O seu formato é typedef tipo novo_tipo; . Assim, um novo tipo de dados é criado com base em um tipo existente, e pode ser usado na declaração de variáveis. Por exemplo, o trecho de programa a seguir cria um novo tipo de dados chamado tipoCodigo que corresponde a um número inteiro e declara a variável codFunc deste tipo.

```
...
typedef int tipoCodigo;
tipoCodigo codFunc;
...
```

O uso de typedef simplifica a declaração de variáveis do tipo estrutura. No trecho de programa a seguir, funcionario é um novo tipo de dados que corresponde a uma estrutura (com os campos codFunc, idadeFunc, sexoFunc e salFunc). Note que funcionario não é uma variável, e sim o nome de um novo tipo de dados. A declaração de variáveis do tipo estrutura pode ser realizada com base neste novo tipo de dados, sem o uso da palavra struct. Isto melhora a legibilidade do programa.

```
...
typedef struct
{
int codFunc; // código do funcionário
int idadeFunc; // idade do funcionário
char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
float salFunc; // salário do funcionário
} funcionario;
...
funcionario funcionario1, funcionario2;
```

Estruturas aninhadas

Uma estrutura pode conter campos que também são estruturas. Quando isto ocorre, tem-se uma estrutura aninhada. O trecho de programa a seguir ilustra a declaração da estrutura funcionario, que possui um dos seus campos (i.e., endFunc) como uma estrutura endereco. O trecho também ilustra o acesso ao campo cidade do campo endFunc de funcionario1. Para cada estrutura aninhada, acrescenta-se o uso do operador ponto.

```
...
struct endereco
{
char rua[40];
int numero;
char complemento[10];
char cidade[35];
```

```
char estado[2]; // sigla do estado
char CEP[8];
};
struct funcionario
{
int codFunc; // código do funcionário
int idadeFunc; // idade do funcionário
char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
float salFunc; // salário do funcionário
struct endereco endFunc; //endereço do funcionário
} funcionario1, funcionario2;
...
printf("O funcionario 1 mora na cidade: %s", funcionario1.endFunc.cidade);
...
```

Uso de operadores relacionais com estruturas

O uso de operadores relacionais (tais como, >, == e <=) somente é possível quando aplicado aos campos de uma estrutura. Portanto, estes operadores não podem ser usados com variáveis do tipo estrutura. O trecho a seguir ilustra um uso correto e um uso incorreto de operadores relacionais com estruturas.

```
...
struct funcionario
{
int codFunc; // código do funcionário
int idadeFunc; // idade do funcionário
char sexoFunc; // sexo do funcionário, M (masculino) e F (feminino)
float salFunc; // salário do funcionário
} funcionario1, funcionario2;
...
if (funcionario1.idadeFunc < 18) // CORRETO
{
...
}
...
if (funcionario2 > funcionario1) // INCORRETO - ERRO!
{
...
}
...
```

Campos de bits

Campos de bits consistem em uma adaptação de estruturas para representar bits. Nesta adaptação, cada campo possui um tamanho em bits. Assim, vários campos podem compartilhar um mesmo byte. Portanto, campos de bits permitem melhorar a alocação de espaço de uma estrutura quando os campos podem ser representados por bits. No trecho de programa a seguir, 1 byte é usado para representar um campo de bits chamado portaSerial. Este byte é

compartilhado da seguinte forma: os 3 primeiros bits são usados para acionar a porta serial; o quarto bit é usado para restabelecer a porta; o quinto e o sexto bits são usados para verificar a paridade; e os últimos dois bits restantes não são utilizados. Caso portaSerial fosse declarada como uma estrutura, e assumindo que cada um dos seus campos fosse declarado como um caractere, ela ocuparia 4 bytes ao invés de 1 byte. Note que a declaração de um campo de bits é quase idêntica à declaração de uma estrutura, diferindo apenas no uso de dois pontos e tamanho (i.e., : tamanho) após a definição de cada campo. O uso do modificador unsigned permite aproveitar todos os bits, desde que não é necessário representar o sinal para números negativos.

```
...
struct portaSerial
{
unsigned int acionaPorta: 3; // 3 bits para acionar porta
unsigned int reset: 1; // 1 bit para restabelecer a porta
unsigned int paridade: 2; // bits de paridade
unsigned int vazio: 2; // bits não utilizados
} portaA;
...
portaA = 0xAF; // atribui o valor hexadecimal AF a variável porta
// note o uso de 0x para indicar base 16 (hexadecimal)
// isso corresponde a (10101111)2
// portanto, acionaPorta recebe 101, reset recebe 0,
// paridade recebe 11 e vazio recebe 11
...
```

União

União consiste em uma variação de estruturas. Em uma união, os diversos campos que a compõem compartilham a mesma porção de espaço de memória. Por exemplo, no trecho a seguir, os campos RA, codFunc e CPF compartilham a mesma porção de espaço. O espaço alocado pela união corresponde ao tamanho do seu maior campo. No exemplo corrente, a união pessoa alocará 12 bytes, que serão compartilhados pelos três campos (Figura 19). O uso de união, portanto, permite economia na alocação de espaço de memória. A união é utilizada quando uma entidade do mundo real pode assumir diferentes significados no programa. Por exemplo, uma pessoa pode ser representada pelo seu RA se ela for um estudante, ou pode ser representada pelo seu código se ela for um funcionário ou ainda pode ser representada pelo seu CPF caso ela não seja estudante ou funcionário. No trecho a seguir, é declarada a variável p1 do tipo união pessoa. Em diferentes partes do programa, esta variável pode estar armazenando dados diferentes, em função do tipo de pessoa. O acesso aos campos de uma união é realizado da mesma forma que o acesso aos campos de uma estrutura, ou seja, por meio do operador de acesso a campos ponto (símbolo .). Note que, após a execução do primeiro comando de atribuição, p1 armazenará o valor 118987623 referente ao código de um funcionário. Após a execução do segundo comando de atribuição, p1 armazenará o valor 3876 referente ao RA de um estudante. Finalmente, p1 receberá o valor 15015015015 após a execução da função strcpy( ), o qual é referente ao campo CPF. Como a mesma porção de memória é compartilhada pela união pessoa, após uma atribuição a um dos campos, o valor atribuído anteriormente a qualquer

outro campo é perdido. Assim, se após a função strcpy( ) for realizada a escrita de p1.RA, um resultado inconsistente será produzido.

Enumeração

Uma enumeração define um conjunto de constantes para representar valores que uma variável pode assumir. Cada constante é representada internamente por um número inteiro, sendo que a primeira constante recebe o valor 0, e as demais são incrementadas de 1 em 1. No trecho de programa a seguir, a enumeração tipo é declarada como o conjunto de constantes motorista (valor 0), secretario (valor 1), docente (valor 2) e outros (valor 3). O campo tipoFunc das variáveis funcionario1 e funcionario2 somente podem assumir as constantes definidas na enumeração. O trecho de programa também ilustra a atribuição de valores ao campo tipoFunc destas variáveis. Note que a atribuição é feita sem o uso de aspas duplas, pois as constantes representam valores inteiros e não strings.

Uso de funções em expressões

Funções que retornam um valor, ou seja, funções que possuem um tipo, podem ser usadas em expressões. O trecho de programa abaixo ilustra o uso de 2 funções em um comando de atribuição. A parte direita deste comando é uma expressão. O corpo das funções é omitido no exemplo. Por outro lado, funções que não retornam um valor, ou seja, funções do tipo void, não podem ser usadas em expressões. O trecho de programa abaixo ilustra o uso incorreto de 2 funções void em um comando de atribuição.

Funções, declarações locais e declarações globais

Em um programa, o escopo de uma variável, ou seja, a abrangência da visibilidade da variável dentro do programa e, por conseguinte, o possível uso de seus valores no programa, depende do local que a variável foi declarada. Uma variável global, que é declarada externamente a função main( ) e também externamente a qualquer função adicional, pode ser usada em qualquer parte do programa. Em outras palavras, uma variável global pode ser usada dentro de qualquer função. Já uma variável declarada dentro de uma função possui escopo local. Assim, esta variável somente pode ser usada dentro do corpo da função e não é visível em outras partes do programa. O trecho de programa a seguir ilustra o uso de variáveis globais e locais e indica um erro no uso de variáveis locais. Quando variáveis em diferentes escopos, por exemplo escopos global e local, possuem o mesmo nome, dentro de uma função é sempre escolhida a variável que possui o escopo local. O trecho de programa a seguir ilustra esta situação.