

# Tópicos avançados

## O ponteiro genérico `void*`

Em C, o tipo `void*` é especial: uma variável desse tipo é um ponteiro genérico, ou seja, um ponteiro para qualquer tipo.

Qualquer ponteiro pode ser convertido implicitamente para `void*`, e vice-versa:

```
int *p;
void *vp;
int x;

p = &x;
vp = p; /* OK: conversão implícita de int* para void* */
vp = &x; /* OK, pelo mesmo motivo (a expressão &x tem tipo int*) */
p = vp; /* OK: conversão implícita de void* para int* */
```

É importante notar que perdemos informação ao converter um ponteiro `T*` (em que `T` é um tipo qualquer) para `void*`: quando temos `T*`, sabemos o endereço em que a variável começa e sabemos o tamanho do tipo `T` (ou seja, `sizeof(T)`). Ao converter esse ponteiro para `void*`, perdemos a informação sobre o tipo original `T`, logo não sabemos mais qual o tamanho. Por essa razão, não podemos dereferenciar, nem incrementar ou decrementar um ponteiro `void*`:

```
void *vp;
int *p;
int x;

vp = &x;
p = &x;

*p = 5; /* OK: copia sizeof(int) bytes a partir do endereço em p */
*vp = 5; /* ERRO! Não sabemos quantos bytes temos que copiar */

p += 2; /* OK: aumenta o valor do endereço de p em 2 * sizeof(int) */
vp += 2; /* ERRO! Não sabemos em qual valor devemos aumentar o endereço */
```

(Nota: alguns compiladores, como o gcc, permitem fazer aritmética com ponteiros `void*`, mas isso é uma extensão não-portável.)

Ponteiros `void*` são usados sempre que se quer uma variável de tipo genérico. Por exemplo, uma implementação genérica de lista duplamente ligada em C poderia ser simplesmente:

```
struct lista {  
    struct lista *next;  
    struct lista *prev;  
    void *data;  
};
```

As funções `malloc` e `calloc` têm, ambas, tipo de retorno `void*`. Normalmente, usamos essas funções para alocar espaço suficiente para N variáveis de algum tipo, usando `sizeof`:

```
struct my_struct *foo = malloc(100 * sizeof(struct my_struct));
```

Porém, nada nos impede de fazer:

```
char *mem = malloc(40); /* aloca 40 bytes */
```

Ou seja, a função `malloc` não tem - e não precisa ter - nenhuma informação sobre o tipo que pretendemos alocar: só importa que a função receba o número total de bytes a ser alocado, e devolva o endereço do começo da região com esse tamanho. Então, como nenhuma informação sobre o tipo é usada, é natural que o tipo de retorno seja `void*`.

Da mesma forma, a função `free` recebe um `void*`: novamente, não importa o tipo do ponteiro que foi alocado, apenas seu endereço.

Nota: dissemos que `void*` é um tipo especial porque, em C,

```
T *p;
```

Declara `p` como um ponteiro para `T`. Seguindo esse padrão,

```
void *p;
```

Deveria declarar um ponteiro para o tipo `void`, o qual não tem nenhum valor - ou seja, um ponteiro para nada. Mas um ponteiro como esse não teria nenhuma utilidade! Logo, decidiu-se usar `void*` como ponteiro genérico em vez disso.

# Ponteiro para função

Funções também ocupam um lugar na memória, e por isso é possível ter ponteiros para funções. Por exemplo:

```
void foo1(int x, int y)
{
    printf("foo1: x = %d, y = %d\n", x, y);
}

void foo2(int x, int y)
{
    printf("foo2: x = %d, y = %d\n", x, y);
}

int main(void)
{
    /* declara um ponteiro para função que recebe dois ints e não retorna nada */
    void (*fptr)(int, int);

    fptr = foo1;
    fptr(1, 2);
    fptr = foo2;
    fptr(3, 4);

    return 0;
}
```

Imprime:

```
foo1: x = 1, y = 2
foo2: x = 3, y = 4
```

Podemos notar, no exemplo acima, que

1. o protótipo da função faz parte do tipo do ponteiro para função; e
2. a chamada a um ponteiro de função tem a mesma sintaxe que uma chamada de função normal.

Na verdade, a mesma relação entre arrays e ponteiros (ambos guardam o endereço de seu primeiro elemento) existe entre uma função "normal" e um ponteiro de função:

```
void foo(int x, int y)
{
    printf("foo: x = %d y = %d\n", x, y);
}

int main(void)
{
    void (*fptr)(int, int);

    fptr = foo;
    printf("foo = %p, fptr = %p\n", foo, fptr);
}
```

Imprime o mesmo endereço para `foo` e para `fptr`.

## Callbacks

Um *callback* é um ponteiro para função que é passado para outra função, a fim de ser chamado numa ocasião específica. Callbacks são muito usados em programação orientada a eventos. Frequentemente, a função que recebe um callback pertence a uma biblioteca, mas o callback em si é escrito pelo usuário. Para se possa ter um callback que lide com tipos definidos pelo usuário - e que não são conhecidos pela biblioteca -, os parâmetros do ponteiro de função devem ter tipo `void*`.

Um exemplo disso é a função `qsort` da biblioteca padrão, que implementa o quicksort. Seu protótipo é:

```
void qsort(void *base, size_t nmem, size_t size,
           int (*compar)(const void *, const void *));
```

Note que a função recebe um ponteiro de função `compar`, que por sua vez recebe dois ponteiros `const void*`. A página de manual de `qsort` explica que

“ O conteúdo da array é ordenado em ordem ascendente, de acordo com a função de comparação apontada por `compar`, que é chamada com dois argumentos que apontam para os objetos sendo comparados. [...] A função de comparação deve retornar um inteiro menor que, igual a, ou maior

que zero se o primeiro argumento é considerado, respectivamente, menor que, igual a ou maior que o segundo.

Então `compar` é o nosso *callback*, e como essa função recebe parâmetros de tipo `const void*`, podemos usá-la com tipos que nós definimos. Graças a isso, a função `qsort` pode ser usada com qualquer tipo definido pelo usuário, sem ter que se preocupar com sua lógica de comparação.

Um exemplo de uso dessa função seria:

```
/* guarda uma array de inteiros e o número de elementos que ela tem */
struct vec {
    int *a;
    size_t elem_num;
};

/* Queremos comparar dois vetores da seguinte forma:
 * v1 é considerado maior que v2 se a soma das entradas de v1
 * for maior que a soma das entradas de v2.
 */

/* NOTA: só chame esta função com ponteiros para struct vec! */
int vec_cmp(const void *p1, const void *p2)
{
    const struct vec *v1 = p1;
    const struct vec *v2 = p2;

    int i, sum1 = 0, sum2 = 0;

    for(i = 0; i < v1->elem_num; i++) {
        sum1 += v1->a[i];
    }
    for(i = 0; i < v2->elem_num; i++) {
        sum2 += v2->a[i];
    }
    /* este valor é maior, igual ou menor que zero
     * se sum1 é menor, igual ou maior que sum2, respectivamente
     */
    return sum1 - sum2;
}
```

```

}

int main(void)
{
    struct vec *v;
    size_t num_elem;
    /* leia v e num_vec de algum lugar (e.g. de um arquivo)
     * de tal modo que v tenha num_elem elementos
     */

    qsort(v, num_elem, sizeof(struct vec), vec_cmp);

    /* faça algo com os vetores em ordem */

    /* libere a memória alocada */
    return 0;
}

```

# Orientação a objetos

É possível emular em C - com certas limitações - *features* presentes em linguagens de programação orientadas a objetos.

Usaremos `struct`s para emular o funcionamento de uma classe.

## Fábrica (Factory) / Construtor

Podemos emular uma fábrica como uma função que retorna um ponteiro alocado dinamicamente para a nossa `struct`.

Por exemplo, poderíamos ter, em um header `vec3.h`:

```

/* um vetor tridimensional (R3). */
struct vec3 {
    double x;
    double y;
    double z;
};

```

```
/* Nota ao usuário: este método deve ter seu valor de retorno dealocado
 * manualmente com free().
 */
struct vec3 *cria_vec3(double x, double y, double z); /* fábrica */
```

E, no arquivo de implementação `vec3.c`:

```
struct vec3 *cria_vec3(double x, double y, double z)
{
    struct vec3 *v = malloc(sizeof(struct vec3));
    if(v == NULL) {
        return NULL;
    }

    v->x = x;
    v->y = y;
    v->z = z;

    return v;
}
```

Podemos também criar algo semelhante a um construtor - uma função que inicializa os campos de uma `struct` recém-criada, e não retorna nada:

```
void init_vec3(double x, double y, double z)
{
    v->x = x;
    v->y = y;
    v->z = z;
}
```

Nesse caso, a responsabilidade de alocar a memória fica com o código do usuário. Além disso, o usuário deve se lembrar de chamar essa função para cada variável desse tipo, após ser criada.

## Método estático

Um método estático é um método que pertence à classe, não a um objeto específico, e por isso não precisa acessar seu ponteiro `this`.

No exemplo anterior, poderíamos adicionar a `vec3.h` o protótipo:

```
/*
 * este método chama cria_vec3() e por isso seu valor de retorno também deve ser
 * manualmente dealocado.
 */
struct vec3 *soma_vec3(struct vec3 *v1, struct vec3 *v2); /* método estático! */
```

E na implementação, em `vec3.c`, teríamos

```
struct vec3 *soma_vec3(struct vec3 *v1, struct vec3 *v2);
{
    return cria_vec3(v1->x + v2->x, v1->y + v2->y, v1->z + v2->z);
}
```

# Declarações estranhas: a Regra da Espiral

Declarações em C podem ser crípticas, especialmente quando envolvem ponteiros de função. Considere:

```
const int (*const foo)[64](int, int);
```

Isso declara a variável `foo` - mas qual o seu tipo? A resposta curta é recorrer a [este site](#). A resposta longa é usar a *Regra da Espiral*, que imita como o compilador analisará a expressão.

Partindo da variável cujo tipo queremos descobrir, percorremos a expressão em espirais de dentro para fora, em sentido horário:

```
const int (*const foo)[64](int, int);
          +++^
```

(Os símbolos `+` indicam a parte da expressão que já analisamos.)

Encontramos um `)`, logo estamos dentro de uma sub-expressão com parênteses, e devemos voltar até o respectivo `(`:

```
((ter algum texto aqui é necessário para preservar o whitespace no começo da linha
abaixo))
```



```

      v-----+++
const int (*const foo)[64](int, int);
      +++^

```

Então `foo` é um ponteiro constante, mas para o que? Percorremos a expressão com uma nova espiral, partindo de toda a sub-expressão que já analisamos:

```

const int (*const foo)[64](int, int);
      ++++++++^

```

Encontramos um `[`, então devemos ir adiante até o respectivo `]`:

```

const int (*const foo)[64](int, int);
      ++++++++^--^

```

Então `foo` é um ponteiro constante para uma array de 64 elementos, mas de que tipo? Outra vez, uma nova espiral:

```

const int (*const foo)[64](int, int);
      ++++++++^

```

Encontramos um `(`, então devemos ir adiante até o respectivo `)`:

```

const int (*const foo)[64](int, int);
      ++++++++^-----^

```

Então `foo` é um ponteiro constante para uma array de 64 ponteiros de função que recebem dois `int`s, mas qual o tipo de retorno?

```

const int (*const foo)[64](int, int);
      ++++++++^

```

Achamos o fim da expressão `(;)`, então voltamos para o começo:

```

vvvvvvvvv- ++++++++^
const int (*const foo)[64](int, int);
      ++++++++^

```

A expressão inteira foi analisada.

Portanto, `foo` é um ponteiro constante para uma array de 64 ponteiros de função que recebem dois `int`s e retornam um `const int`.

# Tipos opacos

`struct`s e `union`s podem ser declarados sem ser definidos:

```
struct foo;
```

Um tipo como esse - declarado mas não definido - é chamado de *tipo opaco*. A princípio, o compilador não sabe o tamanho nem os campos desse tipo, o que gera algumas restrições em seu uso:

```
struct foo;

/* OK: podemos definir ponteiro para tipo opaco,
 * porque todo ponteiro tem o mesmo tamanho
 */
struct foo *ptr;
struct foo var; /* ERRO! Não podemos instanciar tipo opaco */

int main(void)
{
    ptr->stuff = 1; /* ERRO! Não podemos acessar nenhum membro de um tipo opaco */
    *ptr; /* ERRO! Não podemos de-referenciar ponteiro para tipo opaco */
    size_t foo_size = sizeof(struct foo); /* ERRO! Tamanho desconhecido! */
    return 0;
}
```

Com tantas restrições, por que usar um tipo opaco? A resposta: esses tipos permitem obter um certo nível de encapsulamento de dados. Por exemplo, ao criar uma biblioteca, é possível colocar, em um header (`.h`) que será incluído pelo usuário, apenas a declaração do tipo opaco e protótipos de funções que lidam com ponteiros para esse tipo, enquanto a definição do tipo e das funções iria nos arquivos-fonte (`.c`). Com isso, os usuários dessa biblioteca podem manipular os dados do tipo opaco por meio das funções fornecidas, e não precisam se preocupar com os detalhes de implementação do tipo.

Nota: um tipo opaco só pode se usado quando tem uma definição completa em outro arquivo, que deve ser unido (pelo linker) com o primeiro.

# Funções variádicas

Considere as seguintes chamadas a `printf`:

```
printf("foo\n"); /* OK */
printf("%d\n", 3); /* OK */
printf("%f %f %f\n", 1.2, 3.4, 5.6); /* OK */
```

Como é possível que uma mesma função seja chamada com número e tipos de argumentos diferentes?

Isso ocorre porque `printf` é uma função *variádica*. Uma função variádica é indicada pelo uso de reticências (`...`) em seu protótipo; no caso de `printf`,

```
int printf(const char *format, ...);
```

Todos os argumentos antes das reticências têm tipo determinado e são obrigatórios a toda chamada a essa função; a partir das reticências, entende-se que haverá um número arbitrário de argumentos (que pode ser zero), de qualquer tipo.

Logo, as seguintes chamadas estão incorretas:

```
printf(); /* ERRO! */
printf(3.1); /* ERRO! */
printf(1, 2, 3); /* ERRO! */
```

Porque em nenhuma delas o primeiro argumento de `printf` é do tipo `const char*`.

Funções variádicas devem, obrigatoriamente, ter ao menos o primeiro argumento com tipo especificado. Logo, a declaração a seguir *não* é válida:

```
void variadic(...); /* ERRO! Primeiro argumento não especificado */
```

O compilador não tem nenhuma informação sobre os argumentos à função depois das reticências, mas a função em si precisa saber quais argumentos foram passados a ela para que possa usá-los devidamente. No caso de `printf`, essa informação está embutida na string de formatação - por exemplo, se há um `%d`, entende-se que foi passado um inteiro; se há um `%f`, um float, e assim por diante. Porém, o compilador não checa se os argumentos passados a uma função variádica correspondem aos que a função espera, de modo que todas essas chamadas compilam (talvez com warnings):

```
printf("abc\n", 5); /* OK, o segundo argumento não será usado */
printf("%d\n"); /* undefined behaviour, possível falha de segurança */
printf("%f\n", "abcd"); /* tenta imprimir o endereço da string literal como um float */
```

Importante: chamadas do tipo `printf("%d");` em que `printf` espera mais argumentos do que foram passados, constituem uma **falha de segurança**; por isso, não permita que a string de formatação passada para `printf` dependa do input do usuário, a menos que esse input seja sanitizado antes (por exemplo, substituindo todos os `%` por `%%`).

Funções variádicas são implementadas usando-se uma variável do tipo `va_list`, definido no header `stdarg.h`. Essa variável deve ser inicializada pela macro `va_start` e liberada por `va_end`; entre as duas chamadas, podemos usar a macro `va_arg` para extrair um argumento com um dado tipo. Como exemplo, temos uma implementação simplificada de `printf`:

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

/* versão simplificada de printf() que não retorna nada, e entende apenas
 * as formatações %s, %c, %d, %f e %%
 */
void my_printf(const char *fmt, ...)
{
    int in_format = 0;
    va_list args; /* nossa lista de argumentos */

    /* fmt é a última variável conhecida pela função, e deve ser passada a va_start
     * para que a macro saiba onde a lista de argumentos variádicos deve começar
     */
    va_start(args, fmt);

    for(; *fmt; fmt++) {
        if(in_format) {
```

```

in_format = 0; /* todas as nossas formatações têm apenas 1 letra */

switch(*fmt) {
    case '%':
        fputc('%', stdout);
        break;
    case 's':
    {
        /* va_arg extrai um argumento do tipo dado (no caso, char*) e o retorna,
        * de tal modo que a próxima chamada a va_arg retornará o próximo
argumento
        * da lista, e assim por diante.
        * Note que o compilador não pode checar se o tipo do argumento passado
        * é compatível com o que a função espera.
        */
        char *s = va_arg(args, char*);
        fputs(s, stdout);
    }
    break;
    case 'c':
    {
        char c = (char) va_arg(args, int); /* TODO explicar por que extraímos o argumento
como int */
        fputc(c, stdout);
    }
    break;
    case 'd':
    {
        int i = va_arg(args, int);
        /* converta i para string e imprima com fputs() */
    }
    break;
    case 'f':
    {
        double d = va_arg(args, double);
        /* converta d para string e imprima com fputs() */
    }
    break;
    default:

```

```

        /* formatação inválida, ignore. */
    }
} else if(*fmt == '%') {
    in_format = 1;
} else {
    in_format = 0;
    fputc(*fmt, stdout);
}
}
va_end(args); /* terminamos de usar a variável */
}

```

# O operador ,

O operador , avalia seus dois operandos, e tem como resultado o valor do último operando (da direita), descartando o valor do outro. Por exemplo:

```

int main(void)
{
    int w, x, y, z;
    z = (w = 1, x = 2, y = 3);
    printf("w = %d, x = %d, y = %d, z = %d\n", w, x, y, z);
    return 0;
}

```

Imprime

```
w = 1, x = 2, y = 3, z = 3
```

A expressão é avaliada como: `z = ((w = 1, x = 2), y = 3)` -> `z = ((1, 2), 3)` -> `z = (2, 3)` -> `z = 3` -> `3`.

Um uso comum desse operador é em laços `for` que têm mais de um índice:

```

/* suponha que temos uma array quadrada arr de tamanho arrsize */
double sum_diag = 0;
int i, j;

/* note o uso do operador ',' para manipular os dois índices ao mesmo tempo */

```

```
for(i = 0, j = 0; i < arrsize && j < arrsize; i++, j++) {  
    sum_diag += arr[i][j];  
}  
printf("soma das entradas da diagonal principal: %g\n", sum_diag);
```

Por ser conta-intuitivo, esse operador pode ser usado para obfuscação de código.

# Inline assembly

TODO

---

Revision #63

Created Mon, Jan 28, 2019 11:41 PM by Luana

Updated Sun, Mar 17, 2019 8:44 PM by Luana