Funções

Ao contrário de linguagens como Python, em C, quase todo código deve estar dentro de alguma função. Só declarações (de variáveis, tipos, funções, etc) e definições de função podem estar fora de uma função.
Uma função é definida como

```
tipo_de_retorno nome_da_funcao(tipo1 arg1, tipo2 arg2, ..., tipoN argN)
{
  /* comandos */
}
```

Por exemplo:

```
double soma(double a, double b)
{
  return a + b;
}
```

Para chamar a função:

```
double a, b;
/* leia a e b de algum lugar */
double x = soma(a, b);
```

Assim que a função atinge o `return`, passa-se imediatamete o controle da execução para a função que a chamou - ou seja, nada depois do `return` é executado. Por isso, a função

```
double soma(double a, double b)
{
  return a + b;
  printf("somei\n");
}
```

Não imprime "somei".

A função pode ter `void` na lista de argumentos, o que especifica que ela não recebe nenhum argumento:

```

```

```
int funcao_sem_parametros(void)
{
  /* ... */
  return 42;
}
```

A função também pode ter tipo de retorno `void` - nesse caso, a função não retorna nenhum valor:

```
void print_stuff(void)
{
  printf("stuff\n");
}
```

O `return` pode ser omitido em funções com tipo de retorno `void`, mas ainda é possível usá-lo para sair da função prematuramente em caso de erro:

```
void aloca_e_faz_algo(void)
{
  int *ptr = malloc(100 * sizeof(int));
  if(ptr == NULL) {
    return; /* note que o return é usado sem nenhum valor */
  }
  /* faça algo (se estamos aqui, ptr não é NULL) */
  free(ptr);
}
```

Nota: é comum ver implementações desse tipo feitas por programadores novatos:

```
int eh_positivo(int x)
{
  if(x > 0) {
    return 1; /* (A) */
  } else {
    return 0;
  }
}
```

Isso funciona, mas há duas coisas desnecssárias. A primeira é o `else`: se a função não retornar em (A), isso necessariamente significa que a condição `x > 0` é falsa, logo podemos escrever

```
int eh_positivo(int x)
```

```
{
  if(x > 0) {
    return 1;
  }
  return 0;
}
```

A segunda é que tudo isso pode ser escrito simplesmente como

```
int eh_positivo(int x)
{
  return x > 0;
}
```

Porque, para entrar no `if` em (A), a condição `x > 0` já deve ser verdadeira (ou seja, valer 1), e para não entrar no `if`, a condição deve valer 0, que são justamente os valores que retornamos. Logo podemos retornar o valor da expressão diretamente.

Protótipo (declaração) de função

Uma função deve ser declarada antes de ser usada.
Por exemplo, o código abaixo está errado:

```
int main(void)
{
  foo(); /* ERRO! */
  return 0;
}

void foo(void)
{
  printf("foo\n");
}
```

Porque a função `foo` não foi declarada antes de sua chamada. A maneira mais simples de resolver isso é invertendo a ordem das definições:

```
int main(void)
{
  foo();
```

```
  return 0;
}

void foo(void)
{
  printf("foo\n");
}
```

Mas poderíamos também declarar a função antes de definí-la, com um protótipo de função:

```
void foo(void); /* declaração (ou protótipo) */

int main(void)
{
  foo();
  return 0;
}

void foo(void)
{
  printf("foo\n");
}
```

O número e o tipo dos argumentos deve ser o mesmo no protótipo e na definição da função.
O uso mais comum de protótipos é em cabeçalhos (headers). Colocar um protótipo de uma função no header permite que qualquer arquivo que inclua aquele header possa usar a função.
Além disso, o uso de protótipos é necessário quando se tem funções mutuamente recursivas (que chamam uma à outra e vice-versa):

```

/* conta quantos bits 0 e 1 há em um número */
int zeroes = 0;
int ones = 0;

void count_ones(int x); /* protótipo necessário para a chamada abaixo */

void count_zeroes(int x)
{
```

```
  if(x == 0) {
    return;
  }
  if(!(x & 0x1)) {
    zeroes++;
    x >>= 1;
  }
  count_ones(x);
}

void count_ones(int x)
{
  if(x == 0) {
    return;
  }
  if(x & 0x1) {
    ones++;
    x >>= 1;
  }
  count_zeroes(x);
}
```

Ponteiros como argumentos

Os argumentos passados para uma função são, na verdade, cópias do valor original: uma alteração no argumento não se propaga para a função que chama. Por exemplo,

```
void incr(int x)
{
  x++;
}

int main(void)
{
  int x = 5;
  incr(x);
  printf("%d\n", x);
  return 0;
```

```
}
```

Imprime 5, porque a variável `x` da função `incr` não é a mesma que a variável `x` da função `main`, elas apenas tem o mesmo valor no começo de `incr`. Se quisermos que essa alteração se propague, devemos fazer com que a função receba um ponteiro:

```
void incr(int *x)
{
  (*x)++; /* (A) */
}

int main(void)
{
  int x = 5;
  incr(&x);
  printf("%d\n", x);
  return 0;
}
```

Imprime 6.
(Note o uso de parênteses em (A): o operador `++` tem precedencia maior que `*`, de modo que, se não usássemos parênteses, a expressão seria equivalente a `*(x++);`)

Funções não podem receber arrays. É possível escrever o argumento de uma função como

```
void funcao(int arg[])
{
  /* ... */
}
```

Mas `arg` **não** é uma array, e sim um ponteiro. Essa notação é usada simplesmente para mostrar que espera-se que a função seja chamada com uma array. Quando a array é passada para uma função, ela é automaticamente convertida em ponteiro:

```
void func(int arg[])
{
  printf("tamanho em func = %d\n", sizeof(arg));
}

int main(void)
```

```
{
  int arr[20];
  printf("tamanho em main = %d\n", sizeof(arr));
  func(arr);
  return 0;
}
```

Imprime tamanhos diferentes: o tamanho impresso em `func` é igual a `sizeof(int*)`. Se a função tiver que lidar com o tamanho (ou o número de elementos) da array, é preciso passar esse valor como argumento.

Nota: é possível também declarar um parâmetro de função como

```
void func(int arg[20])
{
  /* ... */
}
```

Ainda assim, arg é um ponteiro (como podemos averiguar imprimindo `sizeof(arg)` como no exemplo anterior). O tamanho 20 é simplesmente ignorado pelo compilador. Isso pode confundir quem for ler o seu código - por isso, evite esse tipo de declaração.

Ponteiros como tipo de retorno

Uma função pode também retornar um ponteiro. Porém, **não** se pode retornar ponteiros para váriaveis locais da função. Por exemplo, supondo que temos a seguinte definição de `struct`:

```
struct ponto {
  int x;
  int y;
  int z;
};
```

Então essa função pode parecer, à primeira vista, uma boa forma de criar uma variável desse tipo:

```
struct ponto * cria_ponto(int x, int y, int z)
{
  struct ponto p;

  p.x = x;
  p.y = y;
```

```
  p.z = z;

  return &p; /* ERRO! */
}
```

O problema: como `p` foi declarada no escopo daquela função, assim que a função retornar, `p` vai sair de escopo imediatamente depois do `return`, e não há nenhuma garantia do que passará a ser guardado naquele endereço.
(Na prática, variáveis locais costumam ser armazenadas em uma pilha, de modo que, quando a função que chamou `criaPonto` chamar outras funções, as variáveis locais dessas funções passarão a ocupar o endereço que antes era de `p`.) Portanto, uma função que retorna um ponteiro para uma variável nova deve alocar essa variável dinamicamente:

```
/* esta funcao retorna um endereco obtido com malloc(): lembre-se de chamar free()! */
struct ponto * cria_ponto(int x, int y, int z)
{
  struct ponto *p = malloc(sizeof(struct ponto));

  p->x = x;
  p->y = y;
  p->z = z;

  return p; /* OK! */
}
```

Outra alternativa seria receber um ponteiro em vez de retornar um, deixando a criação da variável a cargo de quem for usar a função:

```
/* note como mudamos de "cria" para "inicializa" para refletir o que a função faz */
void inicializa_ponto(struct ponto *p, int x, int y, int z)
{
  if(p == NULL) {
    return;
  }
  p->x = x;
  p->y = y;
  p->z = z;
}
```

Nota: uma função que retorna um ponteiro alocado dinamicamente deve deixar esse fato documentado, para que a pessoa que for usar essa função saiba que deve chamar `free()` depois.

(De maneira geral, a pessoa que for usar suas funções não deve ser forçada a ler a implementação delas.)

`main`

A primeira função chamada num código em C é `main`. Essa função pode ser definida de duas maneiras:

```
int main(void)
{
  /* (A) */
}
```

Ou

```
int main(int argc, char **argv)
{
  /* (B) */
}
```

Declarando `main` como em (B), essa função pode acessar os parâmetros passados ao programa pela linha de comando: `argc` (argument count) é o número de argumentos, e `argv` (argument values) contém as strings que foram passadas em `argv[0]`, `argv[1]`, ..., `argv[argc - 1]`.
Por exemplo, se o progama chama-se `prog` e é invocado pelo bash como

```
$ ./prog foo bar baz
```

Então teremos `argc == 4`, (o nome do programa é contado), e:

- `argv[0]` vale `"./prog"` (o nome do programa é o primeiro argumento)
- `argv[1]` vale `"foo"`
- `argv[2]` vale `"bar"`
- `argv[3]` vale `"baz"`

Naturalmente, se `main` for declarada como em (A), não é possível acessar os parâmetros da linha de comando.

Tanto em (A) quanto em (B), `main` possui tipo de retorno `int`. A função `main` deve retornar `0` se a execução correu como esperado, ou algum código de erro que não valha zero, caso contrário. Esse valor é chamado de "exit status" do programa.

Revision #3
Created Mon, Jan 28, 2019 11:37 PM by Luana
Updated Mon, Jan 28, 2019 11:58 PM by

Luana