Gambiarras (18+)

Orientação a objetos: Parte 2

Herança / Composição

Essencialmente, é possível forjar uma herança entre duas classes das seguintes maneiras:

1. Copiando-se todos os campos da classe mãe para a classe filha;
2. Usando-se composição em vez disso, colocando-se uma instância da classe mãe na classe filha
3. Também com composição, mas usando-se um ponteiro para a classe mãe, que deve ser alocado quando a filha for instanciada.

Todos os métodos têm suas desvantagens: em (1), uma mudança feita na classe mãe não se propaga automaticamente para a filha; em (2), qualquer tipo de Fábrica que pretendemos usar deve saber quais os campos da classe mãe para inicializá-la, e novamente isso pode mudar; em (3), podemos reutilizar a fábrica da classe mãe, mas temos que garantir que o ponteiro para a classe mãe será alocado quando o objeto da classe filha for criada (a maneira mais fácil seria criar uma fábrica para a classe filha).

Encapsulamento

Podemos obter uma `struct` em que alguns elementos são conhecidos pelo cliente e outros não, semelhante a campos `public` e `private` - usando-se uma `struct` cujos campos são públicos dentro de outra cujos campos são privados. Considere, por exemplo, este header `foo.h`:

```
struct foo {
  int pub; /* campo público */
};

/* funções públicas para manipular struct foo */
struct foo *create_foo(int pub); /* NÃO destrua com free(), use destroy_foo() abaixo! */
int increment_private(struct foo*);
void destroy_foo(struct foo *fpub);
```

E este arquivo de implementação `foo.c`:

```
#include <stdlib.h>
#include <stddef.h> /* offsetof() */
#include "foo.h"

/* tipo privado: não é acessível ao usuário (porque não está no header) */
struct foo_private {
  int priv;
  struct foo foo_public;
};

struct foo *create_foo(int pub)
{
  /* note que alocamos espaço para foo_private, não apenas foo */
  struct foo_private *fpriv = malloc(sizeof(struct foo_private));
  fpriv->priv = 0;
  fpriv->foo_public.pub = pub;

  /* pode parecer que perderemos o ponteiro que alocamos quando a função acabar,
   * mas ainda é possível acessá-lo usando-se offsetof()
   */
  return &fpriv->foo_public;
}

int increment_private(struct foo *fpub)
{
  /* primeiro, temos que fazer cast para o tipo privado para poder acessar seus campos.
   * A macro offsetof calcula o tamanho em bytes entre o começo de uma struct e
   * a posição de algum de seus campos. Com isso e com fpub, podemos recuperar
   * o endereço da struct foo_private.
   * Note o cast para void*: isso impede que seja gerado um endereço errado em função do
   * tipo do ponteiro.
   */
  struct foo_private *fpriv = ((void*)fpub - offsetof(struct foo_private, foo_public));
  (fpriv->priv)++;
}

void destroy_foo(struct foo *fpub)
{
```

```
  /* como alocamos um foo_private, é isso que devemos desalocar. */
  struct foo_private *fpriv = ((void*)fpub - offsetof(struct foo_private, foo_public));
  free(fpriv);
}
```

Métodos não-estáticos

TODO com encapsulamento (o ponteiro this é privado) e ponteiros para funções que recebem this como parâmetro

Templates

(Obrigado ao Vitor Silva por me ensinar esse truque!)

Templates são tipos genéricos em C++. É possível imitar o comportamento dos templates - mas sem type safety - usando o preprocessador. Por exemplo, podemos criar uma lista genérica em um header `list.h` como:

```
#ifndef LIST_H
#define LIST_H

#include <stdlib.h>

#define LIST(X) struct list_##X

#define DEF_LIST(X) \
    LIST(X) { \
        X elem; \
        LIST(X) *next; \
    }; \
    LIST(X) *create_list_##X(X first) \
    { \
        LIST(X) *ptr = malloc(sizeof(LIST(X))); \
        if (ptr == NULL) { return NULL; } \
        ptr->elem = first; \
        ptr->next = NULL; \
        return ptr; \
    } \
    void destroy_list_##X(LIST(X) *ptr) \
```

```
    { \
        while (ptr != NULL) { \
            LIST(X) *next = ptr->next; \
            free(ptr); \
            ptr = next; \
        } \
    } \
    int insert_list_##X(LIST(X) *list, X elem) \
    { \
        LIST(X) *cell = malloc(sizeof(LIST(X))); \
        if (cell == NULL) { return 0; } \
        cell->elem = elem; \
        cell->next = list->next; \
        list->next = cell; \
        return 1; \
    }

#endif
```

Note o uso extenso de `##` nas macros.

Podemos testar nosso "template" com o seguinte arquivo:

```
#include <stdio.h>
#include "list.h"

/* instancia o "template" de lista de int */
DEF_LIST(int)

int main(int argc, char **argv)
{
    LIST(int) *list = create_list_int(42);

    insert_list_int(list, -12);
    insert_list_int(list, 35);

    LIST(int) *cur;
    for(cur = list; cur != NULL; cur = cur->next) {
        printf("%d\n", cur->elem);
```

```
    }

    destroy_list_int(list);
    return 0;
}
```

Funcional: o que dá para fazer

- Funções de ordem alta: uma função de ordem alta (high-order function) é uma função que recebe ou retorna outra função. Funções desse tipo são possíveis em C graças aos ponteiros de função.
- map-reduce: TODO (implementar)

Funcional: o que não dá para fazer

- Currying (aplicação parcial de função): a aplicação parcial de função não é possível em C, porque para isso é necessário que haja algum meio de criar funções em tempo de execução, que não existe nessa linguagem.
- `eval()`: a função `eval()` interpreta a string dada como código na linguagem usada. Como isso é feito em tempo de execução, o compilador de C não poderia ajudar nessa tarefa. Por isso, uma implementação de `eval()` teria que trazer consigo metade do código de um compilador - para analisar a string dada, criar uma árvore de sintaxe a partir dela, etc. Portanto, criar um `eval()` é tecnicamente possível, mas impraticável.
- Lambda / Closure: Lambdas (funções anônimas com captura de variáveis) não são possíveis em C nativo, mas há uma extensão da linguagem para isso.

Shellcode em uma variável

É possível executar os bytes de uma variável como instruções em linguagem de máquina, fazendo um *cast* do tipo da variável para um ponteiro de função, e chamando-o em seguida:

```
#include <stdio.h>

int main(void) {
  /*
   * Shellcode para Linux x86-64:
```

```
   *      mov rax, 60 ; sys_exit
   *      xor rdi, rdi
   *      syscall
   */
  unsigned char shellcode[] = {0xb8, 0x3c, 0x00, 0x00, 0x00, 0x48, 0x31, 0xff, 0x0f,
0x05};

  printf("antes do shellcode\n");
  ((void(*)(void)) shellcode)(); /* executa os bytes em shellcode como instruções */
  printf("depois do shellcode\n"); /* nunca é impresso */
  return 0;
}
```

Por padrão, os executáveis em Linux não permitem que se executem valores na pilha como instruções; para fazer isso, é preciso passar a flag `-zexecstack` para o `gcc`. Do contrário, o código acima gera um Segmentation fault.

Revision #31
Created Sun, Mar 17, 2019 7:20 PM by Luana
Updated Fri, Apr 5, 2019 3:24 AM by Luana