O diário de um programador de merda!

Pedreiro Digital

```
/*
* ----------------------------------------------------------------------------
* "THE BEER-WARE LICENSE" (Revision 42):
* <phk@FreeBSD.ORG> wrote this file. As long as you retain this notice you
* can do whatever you want with this stuff. If we meet some day, and you think
* this stuff is worth it, you can buy me a beer in return Poul-Henning Kamp
* ----------------------------------------------------------------------------
*/
```

Contents

1 — Introdução

Todas as vezes que eu leio um livro a parte que eu mais detesto é a introdução, então eu recomendo fortemente que você, meu amigo leitor, vá direto ao próximo capítulo. Porém, entretando, todavia, se você for um fraco, assim como eu, e gostar de uma introduzida (tcheeee), farei esse agrado para você. Mas antes de mais nada eu gostaria de pedir desculpas pelo conteúdo e linguajar de baixo calão que será utilizado nesse livro. Não sei ainda o que é pior, se são as besteiras, o português ou a linguagem C. As besteiras expressam melhor o meu sentimento (ERM), a linguagem C, porra, essa temos que respeitar, ela é digna de dar o nome do seu filho de *C ANSI*, e o português aprendi com um ano de idade e até hoje eu me considero um *noob*.

1.1 Como contribuir?

É muito fácil contribuir com um projeto de merda, qualquer programador .NET consegue! (brincadeirinha galera!). Deixarei algumas dicas de como ajudar.

- Traduzindo este documento para outras linguas;
- Escrevendo algum capítulo técnico interessante;
- Corrigindo bugs;
- Com idéias;
- Corrigindo erros de português (RÁÁÁ pegadinha do malandro);

1.2 Quem deveria ler este livro?

Sinceramente falando acho que ninguém, mas se você é um gerente de projeto, com tempo livre (nunca vi gerente de projeto ocupado), saia do facebook e venha conhecer um pouco do submundo dos programadores de merda. Além dos gerentes de projetos quem mais pode ler:

- Pessoas com tempo livre;
- Péssimos programadores;
- Programadores java;
- Pessoas que acreditam que seu código é seu filho;

1.3 Quem não deveria ler este livro?

Pessoas que se ofendem facilmente, idosos, cardíacos, gestantes, menores de 18 anos e bons programadores.

1.4 Como este livro funciona?

A idéia desse livro não é ter apenas um ou dois autores, mas vários, qualquer um está livre para colaborar. Cada capítulo do livro é indepentente, ou seja, você poderá encontrar um capítulo sobre um tópico avançado sem ter um capítulo introdutório anterior. Então agora é só baixar o projeto e começar a escrever seu capítulo, em latex.

```
$ git clone https://github.com/patito/DiarioDeUmProgramadorDeMerda.git
```

2 — Reference Counting

"Gerenciamento de memória? Pra que? *Hardware* é tão barato, não vou desalocar a memória". "Sou vida loka, faço pipoca de panela aberta e não libero memória". "Só libera a memória quando quer impressionar a estagiária nova?". Se você pensa dessa maneira, parabéns campeão, você é um dos meus, está liberado para ir assistir *The Big Bang Theory*. Mas para aqueles que são caretas e gostam de viver de forma correta vamos apresentar uma outra solução. O gerenciamento de memória é uma das maiores dores de cabeça dos programadores *C*, fazer essa atividade manualmente com *malloc()* e *free()* funciona, porém se você esquecer um *free()* poderá causar um *memory leak*, se você usar o *free()* de forma errada, você pode corromper seu programa. Essa atividade não precisa ser tão árdua se utilizarmos algumas regras durante o gerenciamento da memória.

2.1 Como funciona?

Reference couting é uma técnica de gerenciamento de memória bem simples, seu funcionamento é baseado em um contador interno do objeto. Esse contador se inicia em 1 (um), quando o objeto é criado. Quando o programador precisar utilizar o objeto ele chama o método *Ref*, assim o contador interno do objeto é incrementado em 1, quando este não for mais utilizado, o método *UnRef* deve ser chamado, o contador interno será decrementado em 1. Quando esse contador interno chegar em zero, quer dizer que ele não é mais referenciado e pode ser liberado. Vamos seguir com um pequeno passo-a-passo.

- person_new | contador = 1 /* Criando o Objeto, contador inicia em 1 */
- person_ref | contador = 2 /* Programador retendo o Objeto, incrementa o contador */
- person_unref | contador = 1 /* Programador liberando o Objeto, decrementa o contador */
- person_unref | contador = 0 /* Objeto não é mais necessário, decrementa o contador */
- Memória será liberada; /* Objeto não é mais referenciado por ninguém e será liberado */

Reparem que chamamos duas vezes o método *person_unref*, um *unref* é para o método *person_new* e outro porque utilizamos o método *person_ref* para reter o objeto.

2.2 Exemplo: Objeto Person

Para exemplificarmos a técnica de *reference couting*, iremos dar o exemplo de um objeto *Person* que contém **nome**, **sobrenome** e **idade**. Esse exemplo é apenas demonstrativo, portanto não se apeguem a nomes e tratamento de erros, a idéia principal é passar o conceito de *reference counting*. Para esse nosso exemplo iremos implementar os métodos:

- person_new: Responsável por alocar o objeto;
- _person_destroy: É um método estáfico responsável por liberar o objeto (memória);
- person_ref: Método responsável por reter o objeto (incrementa o contador);
- person_unref: Método responsável por liberar o objeto;
- person_print: Método para imprimir as informações de Person;

2.3 Implementando o header

Header? O que é isso? Não é só criar um *main.c* e colocar tudo lá dentro? Isso mesmo campeão, quanto orgulho de ti, mas como nossos leitores são frescos e gostam de coisas mais organizadas iremos criar o arquivo *person.h*, colocar as assinaturas dos métodos e até fazer uns comentários para ninguém sair falando que nosso código não é documentado. Estou me sentindo um programador *java*, vamos fazer um diagrama de estados??? NOOOOTTT.

Listing 2.1: person.h

```
#ifndef _PERSON_H_
#define _PERSON_H_

typedef struct {
    char *first_name;
    char *last_name;
    unsigned int age;
    unsigned ref;
} Person;

/*  Method to create the persong object */
Person* person_new(char *first_name,
                   char *last_name,
                   unsigned int age);

/* Retain the object */
void person_ref(Person *obj);

/* Release the object */
void person_unref(Person *obj);

/* Print object information */
void person_print(Person *obj);


#endif /* _PERSON_H_ */
```

2.4 Implementando o método new

Chegou o momento de implementarmos o método que irá criar o nosso objeto, ou seja, o método responsável por alocar a memória. TESÃOOO, já sinto um friozinho na barriga, OPS! Terminarei de escrever essa parte no banheiro e farei muita FORÇA para sair TUUUDO cerrtoo. Meu caro leitor, antes de continuar a parte técnica, vou me abrir um pouco com vocês, e também acender um fósforo. Vamos falar um pouco de merda, mas é só isso que tem nesse documento, ok ok, tu me entendeu. Existe prazer maior no mundo que dar uma c4g4d4 e programar em *C* ao mesmo tempo? Se existe eu desconheço. Eu particularmente acho estranho as pessoas não falarem muito que programam durante o seu momento no trono, porra é muito bom. Na real c4g4r e programar tem muito em comum, em ambos os casos quando você termina, enche a boca e fala orgulhoso: "Esse é meu filhão!" Voltando para a merda do código reparem que *obj->ref* foi inicializada com 1.

Listing 2.2: person_new()

```
Person *person_new (char *first_name,
                    char *last_name,
                    unsigned int age)
{
        if (NULL == first_name) {
            printf("Invalid first_name!\n");
            return NULL;
        }

        if (NULL == last_name) {
            printf("Invalid last_name!\n");
            return NULL;
        }

        Person *obj = (Person *)malloc(sizeof(Person));
        if (NULL == obj) {
                printf("%s Out of Memory!\n", __FUNCTION__);
                return NULL;
        }

        obj->first_name = strdup(first_name);
        obj->last_name = strdup(last_name);
        obj->age = age;
        obj->ref = 1;   /*  Reference Counting */

        printf("Creating object[%p] Person\n", obj);
        return obj;
}
```

2.5 Método person_ref

Agora vamos implementar o método que irá reter o nosso objeto. Lembre-se que para cada Ref no objeto um UnRef deve ser chamado, caso contrário um leak ocorrerá.

Listing 2.3: person_ref()

```
void person_ref(Person *obj)
{
    if (NULL == obj) {
        printf("Person Obj is NULL");
        return;
    }
    obj->ref++; /*  Incrementando nosso contador */
}
```

2.6 Método UnRef

Lembram para que serve o método UnRef? Pra porra nenhuma não é a resposta correta. Ele é responsável por decrementar o contador interno do objeto e verifica se o objeto ainda é referenciado, caso não seja mais, um método para liberar a memória será chamado.

Listing 2.4: person_unref()

```
void person_unref(Person *obj)
{
    if (NULL == obj) {
        printf("Person Obj is NULL");
        return;
    }

    /*  Decrementa o contador e
     *  verifica se esta igual a 0
     */
    if (--obj->ref == 0) {
        printf("Memory Release obj[%p]\n", obj);
        _person_destroy(obj);
    }
}
```

2.7 Liberando a memória

Quando implementamos o método *person_new()* alocamos algumas informações na memória, e esse é o momento de honrarmos o que temos no meio das pernas, assumirmos nossas responsabilidades e mandarmos as informações para caso do caralho, desculpe pessoal pelo ataque de raiva, estou sem café. Como eu estava dizendo, esse é o momento de liberarmos a memória e vivermos felizes sem *memory leak*.

Listing 2.5: _person_destroy()

```
static void _person_destroy(Person *obj)
{
    if (NULL == obj) {
        printf("Person object is NULL!\n");
        return;
    }

    if (NULL != obj->first_name) {
        free(obj->first_name);
    }

    if (NULL != obj->last_name) {
        free(obj->last_name);
    }

    free(obj);
}
```

2.8 Implementando o main.c

Dentro do *main.c* que iremos utilizar os métodos que implementamos acima. Reparem que para cada método *person_new()* tem um *person_unref()* associado. E você está livre para fazer testes, tente comentar a linha *person_unref(father);* e repare que a memória não será liberada.

Listing 2.6: main.c

```
#include <stdio.h>

#include "person.h"

int main()
{
    Person *father = person_new("Jose", "Rico", 65);
    Person *mother = person_new("Beth", "Perigueti", 21);

    person_print(father);
    person_print(mother);

    person_ref(father);
    person_unref(father);

    /* New method - Unref */
    person_unref(father);
    person_unref(mother);

    return 0;
}
```

2.9 Criando o Makefile

Como eu sou um cara muito legal e to ligado que a grande maioria dos leitores são preguiçosos, irei fazer um *Makefile*, se eu fosse um cara sacana eu faria um *autohell*, hehehe.

Listing 2.7: Makefile

```
CC      := gcc

CFLAGS  := -W -Wall -Werror -I.

BIN     := person

SRC := main.c person.c
OBJ := $(patsubst %.c,%.o,$(SRC))

%.o: %.c
        $(CC) $(CFLAGS) -o $@ -c $<

all: $(OBJ)
        $(CC) $(CFLAGS) -o $(BIN) $(OBJ)

clean:
        $(RM) $(BIN) $(OBJ) *.o $(LIB)
```

2.10 Conclusão

Podemos tirar alguma conclusão boa desse capítulo? O cara que escreveu esse capítulo é um anão, grosso, mal educado e burro. Ok, isso é verdade, mas vamos pular os detalhes. Acredito que o gerenciamento de memória em *C* não precisa ser uma tarefa tão dolorosa, se seguirmos algumas regras básicas podemos tornar essa atividade mais fácil. Esse exemplo é simples, mas a idéia é a mesma para problemas complexos. Lembrando que o código completo está no *github*, na pasta *code*, dentro do capítulo *reference.*