

Orientação a Objetos em C



Cristiano Silva

Introdução

Introdução

Se você tem curiosidade de como aplicar o paradigma de orientação a objetos em uma linguagem imperativa como a linguagem C, você está no lugar certo. Neste eBook, vamos abordar os quatro pilares do paradigma de programação orientada a objetos: Encapsulamento, Herança, Polimorfismo, por fim, Abstração.

Ao longo de 5 capítulos, exploraremos esses conceitos de forma clara e concisa, destacando as vantagens e desvantagens de cada um no contexto da linguagem C. Você terá a oportunidade de entender como estes pilares podem ser implementados de maneira eficiente e prática, mesmo em uma linguagem que não é tradicionalmente orientada a objetos.

Mas este não é um livro comum. Este é um livro vivo. O que isso quer dizer? Ele se estende além destas páginas. Durante o seu desenvolvimento, há recursos utilizados que apresento no meu canal do YouTube, [Faz em C](#). Lá, você poderá encontrar mais materiais sobre a linguagem C.

Então, não perca tempo! Corre lá e se inscreva no canal [Faz em C](#) para acompanhar todas as atualizações e ampliar ainda mais seu aprendizado. Prepare-se para uma jornada de aprendizado que irá expandir suas capacidades de programação e transformar sua compreensão da linguagem C. Vamos começar essa exploração juntos e abrir novas possibilidades no seu caminho como desenvolvedor!

Sumário

Capítulo 1 - Classes

Capítulo 2 - O primeiro pilar - Encapsulamento

Capítulo 3 - O segundo pilar - Herança

Capítulo 4 - O terceiro pilar - Polimorfismo

Capítulo 5 - O quarto pilar - Abstração

Capítulo 1 - Classes

Para trabalhar com o paradigma orientação a objetos, precisamos de classes, não é mesmo? O que é isso? Para que serve isso? Bom, para explicar de uma forma mais ilustrativa, a classe seria mais ou menos assim: imagine que você deseja construir algo, como, por exemplo, uma armadura do Homem de Ferro. Para que isso seja possível, você vai precisar de um projeto para cada parte da armadura, como braços, pernas, capacete e o corpo. E para reunir todas essas partes, haverá outro projeto que é a junção de todos estes outros projetos, obtendo assim o projeto da armadura completa. Bom, o que isso tem a ver com programação orientada a objetos, ainda mais em C?

A tentativa de fazer um paralelo entre a construção da armadura do Homem de Ferro e classes são basicamente a mesma. Para construir um objeto, é necessário um projeto, um esquema de como este objeto será criado em memória. Agora, sua definição seria: a classe é um esquemático de como construir um objeto em memória. Todos os objetos precisam ter um esquemático do que eles vão ter e o que vão fazer. Outro paralelo é a famosa receita de bolo. A receita contém todos os ingredientes e passos necessários para a criação do bolo de chocolate com calda de chocolate. (Não gosto nem um pouco de doces. 😞) Resumindo, a classe é uma receita para objetos. Mas uma pergunta que não quer calar: mas a linguagem C não tem classes? Bom, isto é verdade, mas para contornar essa limitação, temos outra forma de representar uma classe em C, e é através de `struct`.

A `struct` em C é um tipo complexo e personalizado. Com `structs`, podemos agrupar dados que façam sentido para um determinado contexto, podendo ainda ter `structs` dentro de `structs`. Isto é chamado de aninhamento de `structs`.

Problema resolvido? Não! Em classes, podemos chamar os métodos através do objeto instanciado. Em linguagem C, podemos fazer o mesmo? A resposta para isto é: Não! Infelizmente não. A linguagem C não pode chamar métodos através de instâncias de forma direta, associando o objeto com seu método. Bom, até teria uma forma de fazê-lo, mas em certo nível não seria prático; isto traria mais problemas do que soluções. Mas também temos outra forma de contornar essa limitação. Podemos associar o objeto com seus métodos através de um prefixo com o nome da classe. Infelizmente, também não temos namespace em C. Mas o prefixo é muito mais do que o necessário para sanar os problemas.

Criando uma classe

Para ilustrar a criação de uma classe, vamos fazer um comparativo com a linguagem Java para tornar o exemplo mais interessante. A classe para este exemplo será a classe **Employee**.

Em Java, a criação da classe **Employee** fica da seguinte forma:

```
public class Employee {  
  
    private String name;  
    private Integer age;  
  
}
```

O equivalente em C fica da seguinte forma:

```
#define EMPLOYEE_NAME_SIZE 100

typedef struct
{
    char name [EMPLOYEE_NAME_SIZE + 1];
    int age;
} employee_t;
```

Uma diferença básica a se notar é o formato da escrita. O Java usa *Pascal Case* para a nomenclatura das classes e *Camel Case* para nomear métodos e atributos. Ainda não havíamos alinhado a nomenclatura utilizada, mas o momento oportuno seria agora, então vamos lá! Atributos são o mesmo que variáveis, porém fazem parte de um tipo especializado; em outras palavras, atributo é uma variável da classe. Métodos são comportamentos que a classe pode executar, métodos são o mesmo que funções e apenas isso.

Retomando o raciocínio: a escrita em C não tem padronização, mas uma bastante utilizada é o estilo *snake case*. Linguagens como Rust e Python misturam os estilos; cada linguagem tem suas características. Ao longo deste livro, utilizaremos o formato *snake case* para padronizar a escrita.

Em C, podemos criar estruturas da forma já apresentada, utilizando o *typedef*. Isso reduz a escrita no momento de declarar a variável, mas também podemos criar a classe de uma forma mais verbosa da seguinte forma:

```
#define EMPLOYEE_NAME_SIZE 100

struct employee_t
{
    char name [EMPLOYEE_NAME_SIZE + 1];
    int age;
};
```

Nesta forma de definição da estrutura, precisaremos sempre escrever a palavra *struct* para criar a instância, tornando-se um pouco tedioso conforme escrevemos o código e já sabemos que se trata de uma estrutura. Neste livro, sempre usaremos o *typedef* para a criação das estruturas.

Outro ponto é que, em C, precisamos alocar memória de forma estática ou dinâmica. Nos exemplos demonstrados, utilizamos a forma estática, que remove preocupações de gerenciamento de memória. No caso do Java, ele gerencia isso para nós.

Criação da instância

Em Java, para criar uma instância, precisamos de um construtor para preencher o conteúdo das variáveis. Eu sei que poderíamos definir a visibilidade dos atributos como pública, mas na orientação a objetos

precisamos respeitar os pilares. Se fizermos isso, ficaria assim:

```
public class Employee {  
  
    public String name;  
    public Integer age;  
  
}
```

Criar uma classe desta forma tem um problema gravíssimo. Isso viola o pilar do encapsulamento, permitindo que este atributo seja acessado de forma direta. Veremos isso mais adiante no capítulo de encapsulamento. Retomando ao construtor, em Java é necessário fazer uso do construtor para que a instância seja criada de forma consistente, ou melhor dizendo, para criar uma instância válida, pronta para ser usada. Declarando o construtor em Java, o código fica assim:

```
// Employee.java  
public class Employee {  
  
    private String name;  
    private Integer age;  
  
    public Employee (String name, Integer age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public static void main(String[] args) {  
  
        Employee employee = new Employee ("John Doe", 25);  
  
    }  
}
```

Para termos o mesmo comportamento em C podemos fazer da seguinte forma:

```
// employee.c  
#include <string.h>  
#include <stdbool.h>  
  
#define EMPLOYEE_NAME_SIZE 100  
  
typedef struct  
{  
    char name [EMPLOYEE_NAME_SIZE + 1];  
    int age;  
} employee_t;
```

```
bool employee_open (employee_t *object, char *name, int age)
{
    bool status = false;

    if (object != NULL && name != NULL)
    {
        strncpy (object->name, name, EMPLOYEE_NAME_SIZE);
        object->age = age;

        status = true;
    }

    return status;
}

int main (void)
{
    employee_t employee;

    if (employee_open (&employee, "John Doe", 25) == true)
    {
        // Process
    }
}
```

O primeiro argumento deverá ser o objeto que será tratado pela função/método. Note que temos o prefixo com o nome da classe; isso ajuda muito na leitura do código, no rastreamento e evita colisões de nome com outras funções. Estamos trabalhando com uma instância válida, porém vazia, o que torna necessário um mecanismo para garantir que o objeto está válido e pronto para uso. Essa garantia é obtida através do retorno da função. Se o retorno for verdadeiro, a instância foi devidamente inicializada. Esta abordagem é conhecida como programação defensiva, ou, como eu gosto de chamar, programação orientada a erros. Nesta abordagem, é necessário testar todos os retornos para garantir o fluxo de execução do código, evitando assim situações onde erros podem causar um término abrupto da aplicação.

O Tipo Opaco

Na definição dos atributos da classe *Employee* em Java, é possível notar que fizemos uso dos modificadores de acesso. Na classe foi usado o `private`, que não permite que outras classes acessem os atributos além do contexto no qual elas existem, fazendo necessário a criação de métodos de acesso. Em C, não temos modificadores de acesso para proteger os atributos. Para ter o mesmo comportamento em C, precisaremos modularizar o código, separando a classe do seu uso. Para modularizar a implementação precisamos criar um *header* e um *source* para *employee*.

No header deixaremos a definição do nome da estrutura, mas não a definiremos, a sua definição será feita no *source*. Esta forma de declaração é conhecida como *forward declaration*. Isto diz ao compilador que a definição existe em algum lugar. Confie! O header fica da seguinte forma:

```
// employee.h
#ifndef EMPLOYEE_H_
#define EMPLOYEE_H_

#include <stdbool.h>

typedef struct employee_t employee_t;

employee_t *employee_open (char *name, int age);

#endif /* EMPLOYEE_H_ */
```

A assinatura da função mudou de forma sutil. Este formato se assemelha com o construtor Java. A sua implementação no *source* fica:

```
// employee.c
#include <employee.h>
#include <string.h>
#include <stdlib.h>

#define EMPLOYEE_NAME_SIZE 100

struct employee_t
{
    char name [EMPLOYEE_NAME_SIZE + 1];
    int age;
};

employee_t *employee_open (char *name, int age)
{
    employee_t *object = calloc (1, sizeof (struct employee_t));

    if (object != NULL)
    {
        strncpy (object->name, name, EMPLOYEE_NAME_SIZE);
        object->age = age;
    }

    return object;
}
```

A implementação interna fica bem parecida com a abordagem de orientação a erros. O fato de a estrutura estar dentro do *source* garante que nenhuma classe fora deste escopo irá acessá-los. Lembrando que nesta abordagem precisaremos liberar os recursos após terminar de usar o objeto. O seu uso fica:

```
// main.c
#include <employee.h>
#include <stdlib.h>
```



```
int main (void)
{
    employee_t *employee = employee_open ("John Doe", 25);
    if (employee != NULL)
    {
        //process

        free (employee);
    }

    return 0;
}
```

Conclusão

Em C temos duas formas para definir a criação de uma estrutura/classe. A primeira é através da abordagem orientada a erros. Nesta abordagem, a instância é válida porém vazia, e é necessário verificar se a instância está devidamente inicializado através de um mecanismo de checagem de erros, para tal foi utilizado o próprio retorno como tipo booleano. Esta abordagem possui a vantagem de não precisarmos gerenciar a memória, nos poupando de possíveis problemas como vazamentos de memória, em contrapartida os atributos ficam expostos, dando a impressão que podem ser acessados de qualquer lugar. Para utilizar esta forma, é necessário ter disciplina e bastante controle, pois é muito fácil comprometer a implementação.

Na segunda abordagem, protegemos o nosso contexto de acesso direto, forçando o usuário utilizar as funções para manipular o contexto interno. Porém traz consigo o gerenciamento de memória. É muito fácil esquecer de liberar os recursos e acabar causando *memories leaks*. Estes erros são tremendamente problemáticos de se rastrear, eles não acontecem de forma imediata, levando meses ou anos para serem detectados.

Neste capítulo vimos as diferenças e semelhanças entre a linguagem C e a linguagem Java, e como podemos criar classes de forma similar. Vimos também as duas formas de definir classes em C, sendo uma orientada a erros e outra de forma opaca. No próximo capítulo vamos trabalhar este mesmo código para falar sobre o primeiro pilar da orientação a objetos que é o encapsulamento.

Confira a versão em vídeo no canal [Faz em C](#). Aproveita e já dá aquela moral. Se inscreva e dá um like. Vamos difundir a informação.

Capítulo 2 - O primeiro pilar - Encapsulamento

Encapsulamento é o um dos quatro pilares da programação orientada a objetos. O encapsulamento permite esconder detalhes internos e de eles funcionam, além disso, protege os dados de acesso direto, atuando como uma espécie de guardião dos dados da classe. Isso é crucial para a centralização de regras de negócio, evitando que estas regras fiquem espalhadas por todo o código, o que prejudica a manutenção, fazendo que uma pequena mudança nas regras cause alterações em diversos pontos e muita dores de cabeça.

Para ilustrar o encapsulamento, vamos criar um exemplo de uma conta de banco. Primeiro vamos exemplificar com uma classe em Java, lembrando que já vimos com uma classe é construída e como o construtor é utilizado para a criação de uma instância.

```
// Account.java
public class Account {
    private Double balance;
    private String user;

    public Account(String user, Double value) {
        if (value >= 100.0) {
            this.balance += value;
            this.user = user;
        }
    }
}
```

No código em Java apresentado, não é possível acessar os atributos *balance* e *user* fora do escopo da classe. A linguagem Java provê uma forma de controlar a visibilidade dos seus recursos internos através dos modificadores de acesso. Em C isso também é possível, porém vamos apresentar ambas as abordagens já apresentadas no capítulo de classes, que são a abordagem orientada a erros e a abordagem utilizando o tipo opaco.

O código em C utilizando a abordagem orientada a erros fica da seguinte forma:

```
// account.c
#include <stdbool.h>
#include <string.h>

#define ACCOUNT_USER_SIZE 100

typedef struct
{
    double balance;
    char user [ACCOUNT_USER_SIZE + 1];
} account_t;

bool account_open (account_t *object, char *user, double value)
{
```

```
bool status = false;

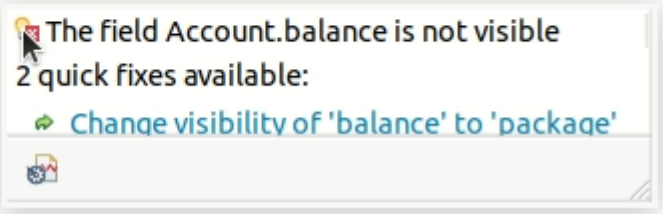
if (object != NULL && user != NULL && value > 100.0)
{
    strncpy (object->user, user, ACCOUNT_USER_SIZE);
    object->balance = balance;

    status = true;
}

return status;
}
```

Neste código temos o seguinte problema: Para o encapsulamento esta forma é muito frágil, aqui é possível acessar qualquer atributo de forma direta. Comparando com o exemplo de uso em Java, quando uma tentativa de acesso direto a um atributo privado obtemos o seguinte erro:

```
1
2 public class TestAccount {
3     public static void main(String[] args) {
4
5         Account account = new Account("John Doe", 101.0);
6
7         account.balance = 10.0;
8     }
9 }
10
```



Não é possível acessar o atributo de fora do escopo. No caso de C podemos acessar e alterar o conteúdo violando a regra do encapsulamento, vejamos no código a seguir:

```
// main.c
int main (void)
{
    account_t account;

    if (account_open (&account, "John Doe", 101.0) == true)
    {
        printf ("Account balance: %.2lf\n", account.balance);
        account.balance = 10.0;
        printf ("Account balance: %.2lf\n", account.balance);
    }
}
```

O *output* gerado com esse código resulta em:

```
Account balance: 101.00
Account balance: 10.00
```

Isto demonstra a fragilidade dessa abordagem em relação ao encapsulamento. Agora seguindo para a próxima abordagem utilizando o tipo opaco. Nesta abordagem, a definição da *struct account_t* será escondida do usuário, como visto no Capítulo 1 - Classes, é necessário modularizar o código para obter o resultado desejado. O arquivo *header* fica com essa cara:

```
// account.h
#ifndef ACCOUNT_H_
#define ACCOUNT_H_

typedef struct account_t account_t;

account_t *account_open (char *user, double balance);

#endif /* ACCOUNT_H_ */
```

No arquivo *source* do *account_t* temos a definição da estrutura e a implementação da função.

```
// account.c
#include <string.h>
#include <stdlib.h>
#include "account.h"

#define ACCOUNT_USER_SIZE 100

struct account_t
{
    double balance;
    char user [ACCOUNT_USER_SIZE + 1];
};

account_t *account_open (char *user, double balance)
{
    account_t *object = calloc (1, sizeof (struct account_t));

    if (object != NULL && user != NULL)
    {
        strncpy (object->user, user, ACCOUNT_USER_SIZE);
        object->balance = balance;
    }

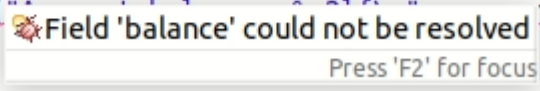
    return object;
}
```

Tentando acessar o atributo fora do escopo do arquivo temos o seguinte erro:

```
#include <stdio.h>
#include <string.h>
#include "account.h"

int main (void)
{
    account_t *account = account_open ("John Doe", 101.0);

    if (account != NULL)
    {
        printf ("Account balance: %.2lf\n", account.balance);
        account.balance = 10.0;
        printf ("Account balance: %.2lf\n", account.balance);
    }
}
```



É possível notar que não podemos acessar mais de forma direta os atributos da estrutura `account_t`. Agora temos realmente o verdadeiro encapsulamento em C, sendo necessário a criação de uma função pública para acessar para alterar o conteúdo interno da estrutura. Isto força o usuário a interagir com o objeto através da API, garantindo que a regra não seja violada.

Conclusão

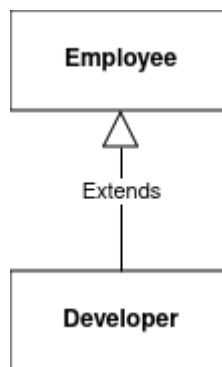
Neste capítulo foi apresentado duas formas de se ter encapsulamento em C. Na abordagem orientada a erros, temos a fragilidade dos atributos, o usuário pode alterar os atributos de forma direta, assim violando as regras de negócio, porém não traz a preocupação do gerenciamento de memória. Na abordagem do tipo opaco, o acesso aos atributos de forma direta são restringidos, o que acaba protegendo e fazendo que as regras sejam sempre aplicadas. Esta abordagem se assemelha a forma que o Java funciona como foi comparado. Entretanto, o gerenciamento de memória fica sobre a responsabilidade do usuário, o que novamente pode trazer problemas de vazamentos de memória. Aqui é um ponto interessante a se destacar: Escolher a abordagem orientada a erros e correr o risco dos atributos serem alterados, causando um estado inconsistente? Ou garantir que as regras sejam respeitadas correndo o risco de ter possíveis vazamentos de memória? É uma *tradeoff*, faça a sua escolha!

Confira a versão em vídeo no canal [Faz em C](#). Aproveita e já dá aquela moral. Se inscreva e dá um like. Vamos difundir a informação.

Capítulo 3 - O segundo pilar - Herança

Herança é um dos 4 pilares da programação orientada a objetos. A herança permite reaproveitar o código e ainda por cima habilita a capacidade de realizar polimorfismo, que será o assunto do próximo capítulo. O reaproveitamento de código se dá pelo fato que uma classe pode herdar as características de uma outra classe. Este relacionamento podemos chamar de relação de mãe e filha, ou pai e filho, classe base e classe derivada. Como podemos ver, é possível nomear esta relação entre as classes da forma que desejar. Para exemplificar este relacionamento, vamos fazer uso do diagrama de classes UML.

O diagrama a seguir apresenta o relacionamento entre a classe mãe **Employee** e as classes filhas **Manager** e **Developer**.



Reusabilidade no ponto de vista do Java

Para uma melhor comparação, vamos implementar em Java e verificar com a implementação em C se equipara. Para a classe **Employee** temos a seguinte implementação:

```
// Employee.java
public class Employee {

    private String name;
    private Integer age;

    public Employee(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName () {
        return this.name;
    }

    public Integer getAge () {
        return this.age;
    }

    @Override
    public String toString() {
        return "Employee name:" + name + ", age:" + age;
    }
}
```

```
}  
}
```

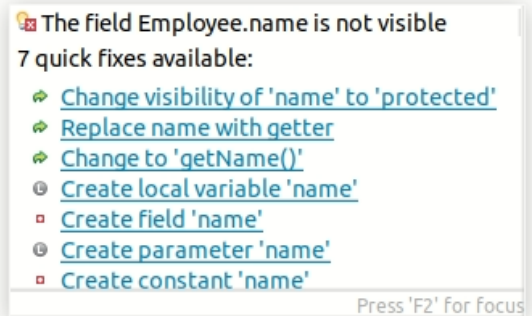
A classe **Employee** é uma classe simples para fins de explicação. A classe contém dois atributos: um nome e a idade. Possui um construtor que precisa de dois parâmetros. Duas funções de `getters`, uma para o nome e outra para a idade, finalizando com uma função `toString`, que formata a mensagem para ser impressa no `stdout`. Para instanciar esta classe, precisamos fornecer um nome e uma idade para o construtor. Os atributos tem visibilidade privada, isto significa que eles serão visíveis somente para a classe **Employee**, o que é um problema para as classes filhas. Para verificar o problema, podemos criar uma nova classe como o nome **Desenvolver**.

```
// Developer.java  
public class Developer extends Employee{  
  
    public Developer(String name, Integer age) {  
        super(name, age);  
    }  
  
    @Override  
    public String toString() {  
        return "Developer name:" + name + ", age:" + age;  
    }  
}
```

A classe **Desenvolvedor** tem uma palavrinha mágica na definição da classe como o nome de `extends` seguida da classe **Employee**. Mas o que isso quer dizer? Aqui está o segredo da herança. Esta linha nos diz o seguinte: A classe **Developer** vai herdar tudo o que a classe **Employee** possuir. Para instanciar a classe **Developer** podemos chamar o construtor da classe **Employee** que já tem a implementação necessário para a inicialização do objeto, e podemos fazer isso através do `super`. Existe outra característica muito interessante aqui. Os métodos podem ser sobrescritos mantendo a assinatura do método, porém com o conteúdo com uma implementação diferente. Isto é o caso do método `toString`. Na classe **Employee** possui o nome **Employee**, e aqui temos o nome de desenvolvedor. E note que não temos as funções `getters`, isto se deve ao fato que já temos a implementação, que é feita pela classe **Employee**, e como **Developer** é filha de **Employee**, logo ela também tem esse método.

Tá! Falou, e falou. Qual era o problema mesmo? Lembra dos atributos da classe **Employee**? Lembra que eles eram privados? Então, na classe **Developer** tentamos utilizar o atributo `name` e o `age` de forma direta. Isto não é possível pois a visibilidade está como `private`. Podemos ver o erro na imagem a seguir:

```
1
2 public class Developer extends Employee{
3
4
5     public Developer(String name, Integer age) {
6         super(name, age);
7     }
8
9     @Override
10    public String toString() {
11        return "Developer name:" + name + ", age:" + age;
12    }
13 }
14
```



Para fazer o acesso a estes atributos precisaríamos usar as funções getters, o que seria uma chatice, certo? Para que a classe filha acesse os atributos de forma direta precisamos alterar de `private` para outra coisa que nos permita fazer este acesso. Mas como podemos fazer isto? Existe outra forma de definir a visibilidade o atributo para que as classes filhas possam ter este acesso, e isto é feito com o `protected`. Existe outro problema para a classe **Employee**. Não é interessante que a classe **Employee** seja instanciada. Para bloquear podemos utilizar a palavrinha `abstract` na definição da classe. Vamos voltar na classe **Employee** e aplicar as alterações:

```
// Employee.java
public abstract class Employee {

    protected String name;
    protected Integer age;

    public Employee(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName () {
        return this.name;
    }

    public Integer getAge () {
        return this.age;
    }

    @Override
    public String toString() {
```



```
        return "Employee name:" + name + ", age:" + age;
    }
}
```

Pronto agora não podemos mais instanciar a classe **Employee**, e ela pode compartilhar os acessos com as classes filhas de forma mas aberta, removendo as burocracias.

Vamos criar uma classe para instanciar um objeto **Developer** e testar as suas chamadas.

```
// TestReuse.java
public class TestReuse {

    public static void main(String[] args) {

        Developer developer = new Developer("John Doe", 27);

        System.out.println("Developer Name: " +
                           developer.getName() +
                           " Age: " +
                           developer.getAge());

        System.out.println(developer.toString());
    }

}
```

Aqui podemos ver claramente que após o objeto desenvolvedor ser instanciado, utilizamos as funções `getName` e `getAge` sem que a classe **Developer** tenha implementado. Isto é o reuso através de herança.

Reusabilidade no ponto de vista do C

Bom agora que temos uma boa noção de como as coisas funcionam em uma linguagem que nos proporciona tais recursos. Vamos voltar ao nosso universo onde as coisas são mais conceituais, e que é preciso exercitar a disciplina para não cair em tentação e tentar dominar este dragão conhecido como ponteiros.

Para replicar o mesmo comportamento em C, é preciso entender uma coisa fundamental: memória. Vamos falar sobre isto depois. Vamos a implementação. Com o código em mãos ficará mais fácil de entender como isto é possível.

Começemos pela classe Funcionário. Lembrando que irei utilizar a abordagem orientado a erros. O resultado do arquivo `employee.h` em C é demonstrado abaixo:

```
// employee.h
#ifndef EMPLOYEE_H_
#define EMPLOYEE_H_

#include <stdbool.h>
```

```
#define EMPLOYEE_NAME_SIZE_ 120

typedef struct
{
    char name [EMPLOYEE_NAME_SIZE_ + 1];
    int age;
} employee_t;

bool employee_create (employee_t *object, char *name, int age);
bool employee_get_name (employee_t *object, char **name);
bool employee_get_age (employee_t *object, int *age);

#endif /* EMPLOYEE_H_ */
```

Como havia dito, estamos usando a abordagem orientada a erros. Para que seja possível verificar se o objeto está válido, precisamos verificar o seu retorno. Dito isso, temos a definição do tamanho do nome do funcionário, temos também a definição da estrutura/classe do Funcionário com o nome e a idade, e três funções, uma para a criação e duas para obter o nome e a idade. Aqui como não temos o recurso de modificador de visibilidade poderíamos ler diretamente, se for necessário alterar alguma coisa no contexto aí é necessário criar uma função para isso, mas para ilustrar o exemplo vamos manter as funções estilo *getters*. Implementando a função no `employee.c`, temos o seguinte resultado:

```
// employee.c
#include "employee.h"
#include <string.h>

bool employee_create (employee_t *object, char *name, int age)
{
    bool status = false;

    if (object != NULL)
    {
        strncpy (object->name, name, EMPLOYEE_NAME_SIZE_);
        object->age = age;

        status = true;
    }

    return status;
}

bool employee_get_name (employee_t *object, char **name)
{
    bool status = false;

    if (object != NULL && name != NULL)
    {
        *name = object->name;
    }
}
```

```
        status = true;
    }

    return status;
}

bool employee_get_age (employee_t *object, int *age)
{
    bool status = false;

    if (object != NULL && age != NULL)
    {
        *age = object->age;

        status = true;
    }

    return status;
}
```

A implementação é simples. Na função `create`, verificamos se a instância está válida, então copiamos o nome e a idade para o contexto, alteramos o status para `true` e retornamos. O mesmo para a função `employee_get_name`, porém obtemos o nome, e finalmente a função `employee_get_age` para obter a idade. Definida a nossa classe base, vamos prosseguir e criar a estrutura/classe do **developer_t**.

```
// developer.h
#ifndef DEVELOPER_H_
#define DEVELOPER_H_

#include "employee.h"

typedef struct
{
    employee_t employee;
    double bonus;
} developer_t;

bool developer_create (developer_t *object, char *name, int age, double
bonus);
bool developer_get_name (developer_t *object, char **name);
bool developer_get_age (developer_t *object, int *age);
bool developer_get_bonus (developer_t *object, double *bonus);

#endif /* DEVELOPER_H_ */
```

Aqui temos um ponto muito importante. Na definição da estrutura/classe do **developer_t** para que haja a herança de fato, é imprescindível que o primeiro atributo seja a estrutura/classe que queremos herdar, que no caso é a estrutura/classe **employee_t**. Como a linguagem não tem a mesma capacidade de

sobrescrever o conteúdo do método igual ao Java, para contornar essa deficiência escrevemos a função com o mesmo nome porém o prefixo referente a estrutura/classe que herda. Para ficar um pouco mais interessante, adicionamos um atributo a mais, o bônus.

Seguinto com a implementação do **developer_t**. O código resultante fica:

```
// developer.c
#include "developer.h"
#include <string.h>

bool developer_create (developer_t *object, char *name, int age, double
bonus)
{
    bool status = false;

    if (object != NULL)
    {
        status = employee_create (object, name, age);
        object->bonus = bonus;
    }

    return status;
}

bool developer_get_name (developer_t *object, char **name)
{
    bool status = false;

    if (object != NULL)
    {
        status = employee_get_name (object, name);
    }

    return status;
}

bool developer_get_age (developer_t *object, int *age)
{
    bool status = false;

    if (object != NULL)
    {
        status = employee_get_age (object, age);
    }

    return status;
}

bool developer_get_bonus (developer_t *object, double *bonus)
{
    bool status = false;
```

```
if (object != NULL && bonus != NULL)
{
    *bonus = object->bonus;

    status = true;
}

return status;
}
```

Vamos analisar a implementação. Na função `create` é possível notar que durante a criação do **developer_t**, para preencher os atributos nome e idade, utilizamos a função `create` do **employee_t**, mas repare o seguinte: Não foi passado como argumento o `employee`, mas sim o desenvolvedor. Estranho não? E o mesmo acontece para as funções `developer_get_name` e `developer_get_age`. O por que isto acontece? Vamos segurar a ansiedade e testar este código. Para isso vamos ao arquivo main, que fica assim:

```
// main.c
#include <stdio.h>
#include "developer.h"

int main(int argc, char **argv) {

    developer_t developer;

    struct
    {
        char *name;
        int age;
        double bonus;
    } data;

    do
    {
        if (developer_create (&developer, "John Doe", 25, 100.0) == false)
            break;

        if (developer_get_name (&developer, &data.name) == false)
            break;

        if (developer_get_age (&developer, &data.age) == false)
            break;

        if (developer_get_bonus (&developer, &data.bonus) == false)
            break;

        printf ("Name: %s, Age: %d, Bonus: %.2lf", data.name, data.age,
data.bonus);
```

```
    } while (false);

    return 0;
}
```

O teste é o seguinte: Instanciamos o **developer_t**, criamos uma espécie de Tupla, através de uma estrutura anônima com o nome de `data`. Utilizamos a função `create` passando os argumentos para preencher o nome, a idade e o bônus. E com o `data` obtemos cada uma das informações utilizadas para a criação do **developer_t**. Para confirmar os dados utilizamos o `printf` para demonstrar a saída. Executando este código temos o seguinte output:

```
Name: John Doe, Age: 25, Bonus: 100.00
```

Agora vamos entender e responder às perguntas realizadas sobre como funciona esta bagaça. Bom para ajudar vamos utilizar o `debug` do eclipse para inspecionar o que está acontecendo em nível de memória.

▶ ➡ &developer	developer_t*	0x7fffffffdc80
----------------	--------------	----------------

Podemos notar na imagem apresentada que o endereço da instância **developer_t** este número 0x7fffffffdc80. Agora vamos expandir um pouco mais e verificar o endereço dos atributos do **developer_t**.

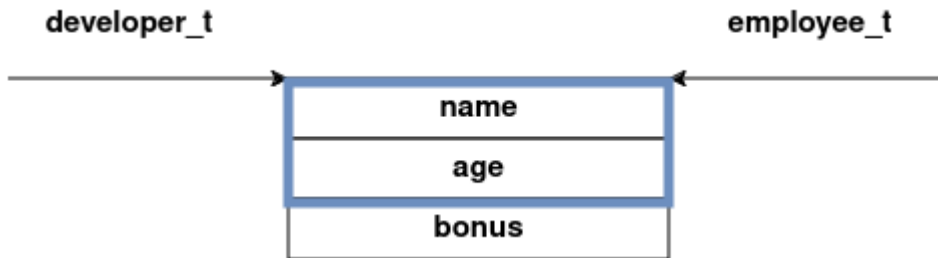
▼ ▶ ➡ &developer	developer_t*	0x7fffffffdc80
▼ 📁 employee	employee_t	{...}
▶ 📁 name	char [121]	0x7fffffffdc80

Podemos notar o seguinte aqui, que o endereço de `name` é o mesmo endereço de **developer_t**. Mas `name` é o primeiro atributo de **employee_t**. Vamos inspecionar o endereço do **employee_t** de forma mais explícita.

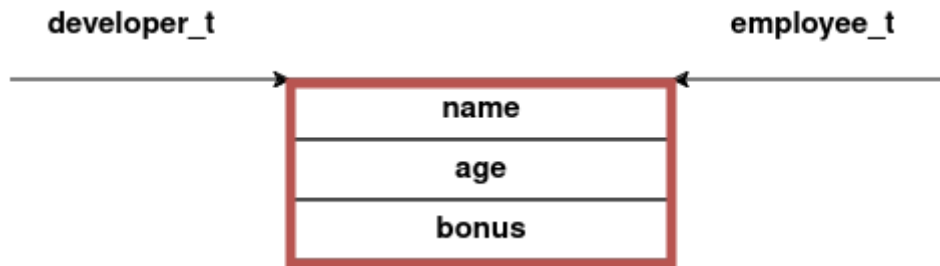
▶ ➡ &developer	developer_t*	0x7fffffffdc80
▶ ➡ &developer.employee	employee_t*	0x7fffffffdc80

O endereço de **employee_t** é o mesmo do seu primeiro atributo, que é o mesmo endereço do **developer_t**? É isto mesmo. A memória alocada para a estrutura é uma única coisa. A forma que acessamos, define a porção de memória que vai ser visualizada através do identificador. Se usarmos o **employee_t** acessaremos somente o início da memória. Se acessarmos via **developer_t** podemos acessar toda ela. Vamos representar isso através de uma imagem para ficar mais fácil de visualizar.

A primeira imagem é a memória do ponto de vista do Funcionário.



A primeira imagem é a memória do ponto de vista do **developer_t**.

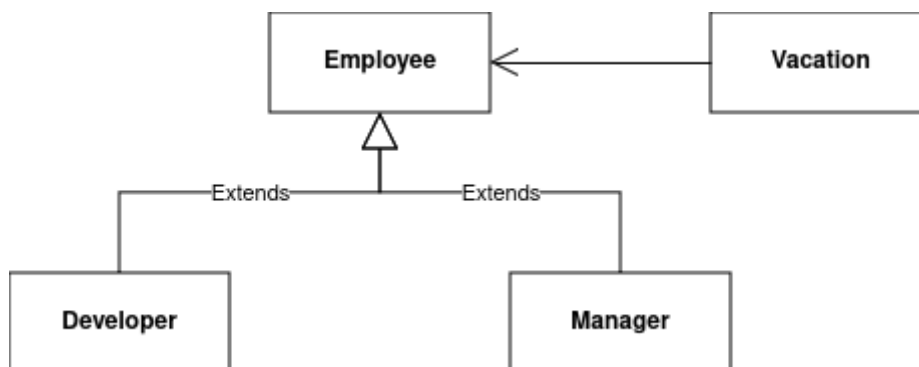


O pulo do gato está aqui. O endereço de memória do primeiro atributo de uma estrutura é o mesmo do seu próprio nome, em outras palavras, temos duas formas de nos referirmos ao mesmo endereço, sob o ponto de vista de **employee_t** temos duas formas: A própria instância e se fizermos um cast para `char *` assim podemos acessar o nome. Sob o ponto de vista do **developer_t** temos três, a própria instância do **developer_t**, através do cast para o Funcionário e também pelo cast de `char *` e todos resultam no endereço da posição onde está o nome. Através desse conhecimento podemos fazer herança por memória em C. Legal não?

Polimorfismo por Herança sob o ponto de vista do Java

Uma outra característica da herança, é que ela permite tratarmos os objetos da mesma forma desde que eles tenham a mesma classe base, ou seja, que herdem a mesma classe. Vamos utilizar o nosso exemplo para explicar o que quero dizer aqui. Até o momento temos a classe **Employee** que é a classe base, e a classe **Developer**. Sabemos que a classe **Developer** é filha da classe **Employee**, isso quer dizer que para qualquer ponto onde se fizer uso de **Employee**, se for usado o **Developer** também funcionará.

Para exemplificar vamos criar mais uma classe com o nome de **Manager**. Esta classe irá herdar o **Employee**. E para demonstrar o polimorfismo por herança, vamos criar uma classe como o nome de **Férias**, que vai fazer uso da classe **Employee**. Como resultado desta nova implementação o diagrama fica da seguinte forma:



A implementação do **Manager** é bem simples, ela somente herda o **Employee** ficando da seguinte forma:

```
// Manager.java
public class Manager extends Employee{

    public Manager(String name, Integer age) {
        super(name, age);
    }
}
```

Para verificarmos o polimorfismo por herança, criamos a classe **Vacation** como mencionado anteriormente, que através de um método estático usa a classe **Employee** como parâmetro para apresentar o nome e o período que o funcionário ficará de férias.

```
// Vacation.java
public class Vacation {

    public static void go_on (Employee employee, Integer days) {

        System.out.println("The employee " + employee.getName() +
                           " will go on vacation for " + days + " days");
    }
}
```

Para testar criaremos outra classe que conterá somente as instâncias do **Developer** e do **Manager**, ambas chamando o método `go_on` da classe **Vacation**.

```
// TestPolymorphismByInheritance.java
public class TestPolymorphismByInheritance {

    public static void main(String[] args) {
        Developer developer = new Developer("John Doe", 25);
        Manager manager = new Manager("Jane Doe", 30);

        Vacation.go_on(manager, 10);
        Vacation.go_on(developer, 30);
    }
}
```

Criamos um desenvolvedor com o nome de `John Doe` com idade de 25 anos, e um gerente `Jane Doe` com 30 anos de idade. Passamos um periodo de 10 dias de férias para o gerente e 30 dias para o desenvolvedor. Executando este código o `output` apresenta a seguinte mensagem:

```
The employee Jane Doe will go on vacation for 10 days
The employee John Doe will go on vacation for 30 days
```


E podemos concluir que: Ambas as instâncias são tratadas como `Employee` pelo simples fato que elas são filhas de `Employee`. E através deste ponto em comum entre as classes **Developer** e **Manager** podemos generalizar comportamentos, alcançando assim, o polimorfismo por herança.

Polimorfismo por Herança sob o ponto de vista do C

Agora que vimos como funciona o polimorfismo em Java em C não difere muito pelo fato que já entendemos como é feita a herança via memória. Primeiro, vamos implementar o **Manager** segundo o diagrama de classes.

```
// manager.h
#ifndef MANAGER_H_
#define MANAGER_H_

#include "employee.h"

typedef struct
{
    employee_t employee;
} manager_t;

bool manager_create (manager_t *object, char *name, int age);
#endif /* MANAGER_H_ */
```

Simplificamos a implementação para somente ficarmos com o que é necessário para demonstrar como funciona, mas segue o mesmo padrão. O `manager_t` tem o `employee_t` como atributo, como bem sabemos, isto é o suficiente para habilitar o polimorfismo por herança. Seguindo com a implementação da função temos:

```
// manager.c
#include "manager.h"
#include <string.h>

bool manager_create (manager_t *object, char *name, int age)
{
    bool status = false;

    if (object != NULL)
    {
        status = employee_create (object, name, age);
    }

    return status;
}
```

No `source` temos somente a implementação da função `manager_create`. Com o `manager_t` criado, vamos implementar a classe **Vacation**. A classe **Vacation** vai conter uma única função similar o que já vimos no Java.

```
// vacation.h
#ifndef VACATION_H_
#define VACATION_H_

#include "employee.h"

void vacation_go_on (employee_t *object, int days);

#endif /* VACATION_H_ */
```

A classe **Vacation** precisa do include `employee.h`, porque o tipo **employee_t** é utilizado como parâmetro na função `vacation_go_on`. A sua implementação apresenta um texto simples utilizando as funções do `employee`.

```
// vacation.c
#include "vacation.h"
#include <stdio.h>

void vacation_go_on (employee_t *object, int days)
{
    char *name;

    if (employee_get_name (object, &name) == true)
    {
        printf ("The employee %s will go on vacation for %d days\n", name,
days);
    }
}
```

E com isso concluímos a implementação. Para testar vamos alterar o main para que demonstre o polimorfismo por herança.

```
// main.c
#include <stdio.h>
#include "developer.h"
#include "manager.h"
#include "vacation.h"

int main(int argc, char **argv) {

    developer_t developer;
    manager_t manager;

    do
    {
        if (developer_create (&developer, "John Doe", 25, 100.0) == false)
            break;
    }
```

```
        if (manager_create (&manager, "Jane Doe", 30) == false)
            break;

        vacation_go_on (&manager, 10);
        vacation_go_on (&developer, 30);

    } while (false);

    return 0;
}
```

No main instanciamos um desenvolvedor e um gerente. Em seguida iniciamos ambas as instâncias, o desenvolvedor com o nome de John Doe, com 25 anos de idade, e o gerente com o nome de Jane Doe com 30 anos de idade. Para o gerente aplicamos 10 dias de férias, e para o desenvolvedor 30 dias de férias. Rodando o programa temos o `output`:

```
The employee Jane Doe will go on vacation for 10 days
The employee John Doe will go on vacation for 30 days
```

Exatamente idêntico a versão de Java. Com esta implementação podemos ver o poder dos ponteiros.

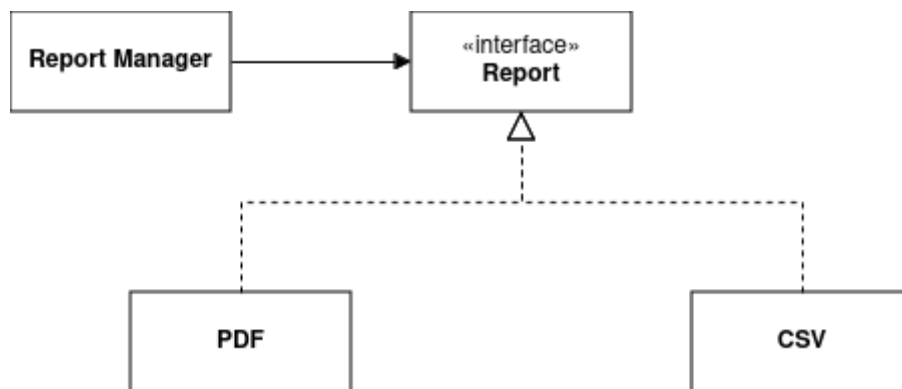
Conclusão

Neste capítulo aprendemos e entendemos o porquê de como a herança funciona em C. Fizemos a comparação com o Java para ter uma visão mais clara do comportamento, e notamos que no Java as coisas são mais orgânicas, pelo fato da linguagem oferecer os mecanismos para tal, o que não é o caso de C. Java o oferece controle de visibilidade para os atributos, não que não possamos ter o mesmo com C, mas precisaríamos ir pela abordagem do tipo opaco. Aliás fica de exercício para você implementar e comparar. O Java também oferece um bloqueio que não permite que uma classe seja instanciada com o `abstract`, novamente não temos isto em C. Porém o ponto mais complicado seria o recurso de `override`, que permite que classes filhas sobrescrevam o comportamento da classe base, o que em C fizemos através do encapsulamento pois não temos como associar as funções, o fato que não temos uma `vtable`. Para contornar precisamos ficar rescrevendo o tempo todo as funções adicionando o prefixo, poderíamos até chamar as funções do **Employee** diretamente, mas acabaria causando confusão. Resumindo, herança em C não é recomendada, dá muito trabalho e é pouco útil. Mas não fique triste para isto temos a composição que funciona exatamente da mesma forma, porém sem a preocupação em manter o primeiro atributo da estrutura. No próximo capítulo vamos ver o Polimorfismo, e para mim é uma das coisas mais legais de se fazer em C.

Confira a versão em vídeo no canal [Faz em C](#). Aproveita e já dá aquela moral. Se inscreva e dá um like. Vamos difundir a informação.

Capítulo 4 - O terceiro pilar - Polimorfismo

Já vimos o polimorfismo por herança no capítulo anterior, mas aqui apresentaremos uma outra forma de alcançar o mesmo resultado. O polimorfismo também pode ser obtido através de contratos. O que significa estes contratos? Contratos são regras pré-estabelecidas no qual quem assinar o contrato precisará seguir as regras a partir do momento da assinatura. Trazendo para o mundo da programação, estes contratos são conhecidos como *interfaces* ou *traits*, o pessoal gosta mesmo de dar nomes as coisas que já tem nomes, não é mesmo? Bom, estas interfaces possuem comportamentos, as classes que tiverem interesse em obter estes comportamentos precisam implementar estes métodos para que realizem uma ação correspondente independentemente da natureza da classe. Em outras palavras, a interface se assemelha à herança, só que ao invés de herdar os atributos e métodos da classe base, só é herdado os métodos que são obrigatórios, mesmo que não tenha código internamente, mas precisa ter. Para entendermos melhor o que é interface. Vamos implementar um relatório que pode imprimir o conteúdo tanto em CSV quanto em PDF. Podemos visualizar o diagrama do exemplo logo abaixo:



Como podemos ver no diagrama teremos uma classe **Report** que é a interface propriamente dita. Logo abaixo, temos duas classes **CSV** e **PDF** que precisam implementar a interface **Report** para poder obter o comportamento necessário para que os relatórios de um determinado formato seja impresso. E para realizar a impressão temos a classe **ReportManager**, que utiliza a interface **Report** para imprimir o conteúdo. Apresentado o nosso projeto, vamos iniciar com o Java para termos uma visão de implementação e posteriormente comparar com a a implementação em C.

Polimorfismo sob a ótica de Java

Como estamos comparando com o Java, vamos utilizar a nomenclatura que o Java utiliza. No mundo Java estes contratos são conhecidos como interfaces e como já era de se esperar o Java já tem uma palavra reservada para identificar este relacionamento. Ao invés de criarmos uma classe podemos criar uma interface. E é isto que vamos fazer agora, vamos implementat o **Report**. O código fica da seguinte forma:

```
// Report.java
public interface Report {
    String getFormat ();
    String getTitle ();
    String getContent ();
}
```

Nesta interface criamos três métodos para estabelecer o contrato. Então as classes que implementar esta interface precisam obrigatoriamente implementar estes métodos, ou pelo menos ter eles declarados. Esta interface apresenta três métodos: `getFormat`, `getTitle` e `getContent`. Eles devem retornar uma string representando a sua finalidade, e isto é responsabilidade da classe implementadora. Dito isso, mas antes de prosseguirmos repare na declaração e veja que o nome é `interface`, e não mais classe. Como havia dito o Java já tem a palavra reservada para este fim. Legal, não? Até tem uma convenção para identificar o que é classe, mas não gosto de nenhuma delas, então fica ao seu critério pesquisar sobre elas. No meu caso uso a palavra na sua forma pura que no caso é **Report**. Para o primeiro contrato vamos usar o **CSV**, o código fica da seguinte forma:

```
// CSV.java
package polimorfism;

public class CSV implements Report{

    private String title;
    private String content;

    public CSV(String title, String content) {
        this.title = title;
        this.content = content;
    }

    @Override
    public String getFormat() {
        return CSV.class.getSimpleName();
    }

    @Override
    public String getTitle() {

        return this.title;
    }

    @Override
    public String getContent() {
        return this.content;
    }
}
```

Com a classe **CSV** implementada, vamos analisar alguns pontos. Primeiro ponto, note que na definição da classe tem um palavra `implements`, esta palavrinha diz que um contrato com a interface **Report** foi assinado. Após esta definição tem dois atributos, sendo um título e ou outro o conteúdo, e eles são preenchidos na criação do objeto via construtor. Mais abaixo é possível notar os métodos com o mesmo nome que os nomes da interface **Report**. Existe um nome acima de cada um dos métodos com o nome de `Override`, isto indica que a assinatura é a mesma porém o conteúdo foi modificado, serve mais como marcação para saber que este método vem de outro lugar. Repare que os métodos `getTitle` e `getContent` retornam os atributos preenchidos na criação do objeto, e podemos usar o contexto da interface para retornar os atributos, na verdade os seus valores. Vejamos agora a classe **PDF**.

```
// PDF
package polimorfism;

public class PDF implements Report{

    private String title;
    private String content;

    public PDF(String title, String content) {
        this.title = title;
        this.content = content;
    }

    @Override
    public String getFormat() {
        return PDF.class.getSimpleName();
    }

    @Override
    public String getTitle() {

        return this.title;
    }

    @Override
    public String getContent() {
        return this.content;
    }

}
```

A classe **PDF** segue o mesmo estilo da classe **CSV**, só que o seu conteúdo é para um relatório no formato pdf. Para utilizar estes formatos de relatórios, faz-se necessário um gerenciador que utilize a interface. Para este fim vamos implementar a classe **ReportManager**.

```
// ReportManager.java
package polimorfism;

public class ReportManager {

    public static void generate (Report report) {

        System.out.println("Format: " + report.getFormat());
        System.out.println("Title: " + report.getTitle());
        System.out.println("Content: " + report.getContent());
    }

}
```

O **ReportManager** é responsável por executar o algoritmo de geração do relatório. O algoritmo é simples, basicamente apresentar o conteúdo do relatório através de uma chamada encadeada dos métodos. Há um ponto muito interessante aqui: O **ReportManager** não faz qualquer distinção de qual é o formato, ele espera um **Report** e nada mais. Isto nos diz o seguinte: Qualquer coisa que implemente **Report** pode ser utilizada. Mas também não vamos banguçar o rolê, né? Como por exemplo, criar uma classe **Vaca** e passar como relatório! Pelo amor?

Para testar, criaremos outra classe com o nome de **TestReport**, onde teremos um título e um conteúdo que será passado para as instâncias de **CSV** e **PDF**. Assim o **ReportManager** pode gerar os relatórios.

```
// TestReport.java
public class TestReport {

    public static void main(String[] args) {

        String title = "Some Title";
        String content = "Some Content";

        ReportManager.generate(new CSV(title, content));
        ReportManager.generate(new PDF(title, content));
    }
}
```

Executando o teste, a saída fica da seguinte forma:

```
Format: CSV
Title: Some Title
Content: Some Content
Format: PDF
Title: Some Title
Content: Some Content
```

A linguagem Java nos dá todos os recursos necessários para implementar polimorfismo via interfaces de forma transparente. Me dá até vontade de usar a linguagem para fazer as minhas coisas pessoais. Mas é só vontade mesmo! Vamos para o que interessa!!!

Polimorfismo sob a ótica de C

Agora que temos o ponto de vista de como funciona uma implementação de interfaces em Java, a pergunta que não quer calar... Como se faz isto em C? Lembra da forma que fizemos a implementação para herança em C? Pois é, mas agora a perspectiva é outra, com polimorfismo via interface em C é muito mais atrativo que de a herança em si. Vamos seguir implementando a interface **report_base_t**.

```
#ifndef REPORT_BASE_H_
#define REPORT_BASE_H_
```

```
typedef struct
{
    void *object;
    char *(*get_format) (void *object);
    char *(*get_title) (void *object);
    char *(*get_content) (void *object);
} report_base_t;

#endif /* REPORT_BASE_H_ */
```

Esta coisa estranha que você está vendo é o que chamamos de interface em C. Mas não tem nenhuma palavrinha mágica para identificar que é uma interface? Não. Aqui usaremos o sufixo base para indentificar as interfaces. O que é aquele troço ali? Uma variável `void *`? Exatamente. E estas funções entre parênteses e com o asterisco no início? Isto são ponteiros de funções. Não se assuste, vamos ir com um passo de cada vez.

Na linguagem C não temos as palavrinhas mágicas para dar suporte a criação de interfaces, mas isto não nos impede de sermos criativos, e de certa maneira bem técnico. Muitas pessoas consideram esta habilidade como avançada. Se entendeu, não perca tempo pode colocar no currículo que já tem C avançado. Vamos começar com o `void *`, isto é uma variável que podemos considerar genérica, aqui o que importa é o endereço da memória, e não o seu tipo, poderíamos ter usado o `int` que representa o tamanho da palavra do processador, mas tem tipo, então o `void *` deixa claro que intenção é se referir a coisas genéricas. Esta variável tem um propósito especial, vou explicar o por quê. Em Java quando uma classe implementa uma interface, ela meio que se torna o tipo da interface, e com isso ela pode compartilhar o seu contexto com o escopo dos métodos da interface, isto significa que os atributos podem ser utilizados dentro dos métodos como se fossem os métodos da própria classe. Em C isto não acontece, então para que o contexto seja utilizado no escopo das funções que a interface provê, precisamos de uma forma de levar este contexto para dentro destes escopos, é onde entra o `void *object`, ele vai carregar a referência do contexto para dentro do escopo das funções e lá dentro precisaremos recuperar para ter acesso ao contexto.

Agora este monstinho `char *(*get_format) (void *object);`. Isto é um ponteiro de função, é uma definição de como a função deve ser, vamos por partes para entender. Lembra da definição de uma função? Se não vamos relembrar: A função é dividida em três partes sendo elas:

```
<retorno> <nome da função> <paramêtros>
```

Agora aplicando esta estrutura à ponteiros de função fica:

```
<retorno> (*<nome da função>) <paramêtros>
```

Esta notação serve para criar um tipo igual a um inteiro por exemplo. Se criar uma variável do tipo `int` e usar um `float` os números após a casa decimal será descartado. O mesmo se aplica aqui, se criar um tipo ponteiro de função as funções que serão usadas para a atribuição devem respeitar este formato. Em outras palavras é uma variável do tipo ponteiro de função. Logo, logo vamos fazer uso, caso não tenha entendido. Seguindo definimos a primeira classe `csv_t`.


```
#ifndef CSV_H_
#define CSV_H_

#include <stdbool.h>
#include "report_base.h"

typedef struct
{
    report_base_t base;
    char *title;
    char *content;
} csv_t;

bool csv_open (csv_t *object, char *title, char *content);

#endif /* CSV_H_ */
```

Conforme fizemos em herança colocando o atributo na primeira posição da estrutura, o mesmo precisa ser feito aqui para usarmos as interfaces sem precisar especificar o argumento. E adicionamos mais dois atributos o nome e conteúdo. Para a criação da instância, definimos uma função para a criação recebendo os argumentos título e conteúdo. Seguindo com a implementação do csv temos:

```
#include "csv.h"
#include <string.h>

static void init (csv_t *object);
static char *csv_get_format (void *object);
static char *csv_get_title (void *object);
static char *csv_get_content (void *object);

bool csv_open (csv_t *object, char *title, char *content)
{
    bool status = false;

    if (object != NULL)
    {
        init (object);

        object->title = title;
        object->content = content;

        status = true;
    }

    return status;
}

static void init (csv_t *object)
{
    memset (object, 0, sizeof (csv_t));
}
```

```

    object->base.object = object;
    object->base.get_format = csv_get_format;
    object->base.get_title = csv_get_title;
    object->base.get_content = csv_get_content;
}

static char *csv_get_format (void *object)
{
    (void) object;
    return "CSV";
}

static char *csv_get_title (void *object)
{
    csv_t *csv = (csv_t *) object;

    return csv->title;
}

static char *csv_get_content (void *object)
{
    csv_t *csv = (csv_t *) object;

    return csv->content;
}

```

Para satisfazer a interface, criamos 4 funções, sendo todas elas estáticas, isso quer dizer que estas funções só serão acessíveis no escopo do arquivo `csv.c`. A primeira função nomeamos como `init`. Esta função tem o propósito de realizar o preenchimento da interface. Perceba que ela é um ponto muito importante, qualquer vacilo aqui, e a aplicação explode. Veja que a atribuição desta linha `object->base.object = object;` guarda a referência do contexto, isto precisa ser feito caso queira obter o contexto no escopo da função que implementa a interface. No restante atribuímos as funções as variáveis, que são na verdade ponteiros de função. No final tudo é variável, até funções como se pode ver. Na implementação da função `csv_get_title` recuperamos o contexto através do `cast` de `void *` para `csv_t *`. Agora podemos utilizar o contexto e retornar o título atribuído ao contexto na sua criação. O resto é a mesma coisa. A classe `pdf_t` segue o mesmo procedimento.

```

#ifndef PDF_H_
#define PDF_H_

#include <stdbool.h>
#include "report_base.h"

typedef struct
{
    report_base_t base;
    char *title;
    char *content;
} pdf_t;

```

```
bool pdf_open (pdf_t *object, char *title, char *content);

#endif /* PDF_H_ */
```

A mesma coisa para a sua implementação.

```
#include "pdf.h"
#include <string.h>

static void init (pdf_t *object);
static char *pdf_get_format (void *object);
static char *pdf_get_title (void *object);
static char *pdf_get_content (void *object);

bool pdf_open (pdf_t *object, char *title, char *content)
{
    bool status = false;

    if (object != NULL)
    {
        init (object);

        object->title = title;
        object->content = content;

        status = true;
    }

    return status;
}

static void init (pdf_t *object)
{
    memset (object, 0, sizeof (pdf_t));

    object->base.object = object;
    object->base.get_format = pdf_get_format;
    object->base.get_title = pdf_get_title;
    object->base.get_content = pdf_get_content;
}

static char *pdf_get_format (void *object)
{
    (void) object;
    return "PDF";
}

static char *pdf_get_title (void *object)
{

```

```
    pdf_t *pdf = (pdf_t *) object;

    return pdf->title;
}

static char *pdf_get_content (void *object)
{
    pdf_t *pdf = (pdf_t *) object;

    return pdf->content;
}
```

Para utilizar a interface precisamos de um gerenciador. Para este papel temos o `report_manager`, esta classe faz uso do `report_base_t`.

```
#ifndef REPORT_MANAGER_H_
#define REPORT_MANAGER_H_

#include "report_base.h"

void report_manager_generate (report_base_t *base);

#endif /* REPORT_MANAGER_H_ */
```

É uma classe bem simples, note que não é necessário criar nenhum tipo de estrutura, que o caso do Java, apesar de ser um método estático, ainda precisa estar encapsulado em uma classe. E possui somente uma função, que é gerar o relatório.

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

void report_manager_generate (report_base_t *base)
{
    printf ("Format: %s\n", base->get_format (base->object));
    printf ("Title: %s\n", base->get_title (base->object));
    printf ("Content: %s\n", base->get_content (base->object));
}
```

A implementação do `report_manager_generate` é responsável pelo algoritmo de geração do relatório. Esta função utiliza a interface `report_base_t` chamando as funções de forma encadeada, porém as chamadas são através da instância. Lembre que ponteiros de função também são variáveis? Aqui utilizamos este recurso. Lembre do `void *object`? Olha ele sendo passado via argumento para as funções. A forma que montamos a interface precisa carregar toda a informação que ela contém e agora podemos como ele chega no contexto das classes `pdf_t` e `csv_t`.

Finalmente, vamos testar para ver se realmente funciona.

```
#include "report_manager.h"
#include "csv.h"
#include "pdf.h"

int main(int argc, char **argv)
{
    csv_t csv;
    pdf_t pdf;

    char *title = "Some Title";
    char *content = "Some Content";

    do
    {
        if (pdf_open (&pdf, title, content) == false)
        {
            break;
        }

        if (csv_open (&csv, title, content) == false)
        {
            break;
        }

        report_manager_generate (&pdf);
        report_manager_generate (&csv);

    } while (false);

    return 0;
}
```

O teste é bem simples, criamos duas instâncias uma para cada formato de relatório, CSV e PDF. Duas variáveis do tipo ponteiro são criadas sendo uma para o título e outra para o conteúdo. Iniciamos as instâncias testando o retorno, como já disse que uso a abordagem orientada a erros. Caso a inicialização ocorreu de forma satisfatória, realizamos a chamada da função `report_manager_generate` passando o pdf, em seguida o csv. E podemos ver no `output` o resultado da implementação.

```
Format: PDF
Title: Some Title
Content: Some Content
Format: CSV
Title: Some Title
Content: Some Content
```

E obtemos o mesmo resultado, mas com gosto de vitória. Depois disto não tem mais volta, todo o mundo que você compreende se torna pequeno, e você começa a vislumbrar possibilidades e realizações. Bom este

não foi o meu caso, precisei treinar bastante isto de forma exaustiva para poder ganhar fluência sobre como manipular e entender estas coisas.

Conclusão

Neste capítulo abordamos o polimorfismo. Fizemos uma comparação com o Java e aplicamos diversas técnicas de ponteiros utilizando o conhecimento adquirido no capítulo de Herança sobre como memória funciona. O mesmo foi aplicado aqui, mas com um contexto um levemente diferente. O polimorfismo é uma ótima forma de implementar de forma flexível e extensível. Este é um recurso que utilizo para criar algoritmos e proteger o domínio da aplicação contra tecnologia através de injeção de dependências, que diferente da herança que já traz comportamentos e características já implementadas. No próximo capítulo iremos abordar a abstração.

Confira a versão em vídeo no canal [Faz em C](#). Aproveita e já dá aquela moral. Se inscreva e dá um like. Vamos difundir a informação.

Capítulo 5 - O quarto pilar - Abstração

A Abstração é o último, e mais importante pilar do paradigma orientação a objetos. A abstração permite extrair idéias fundamentais e isolar essas características, se tornando assim a base para outras coisas. Com a abstração ainda é possível abstrair comportamentos escondendo detalhes internos de uma implementação. Ela reúne todas os outros três pilares e acaba sendo a combinação de todas.

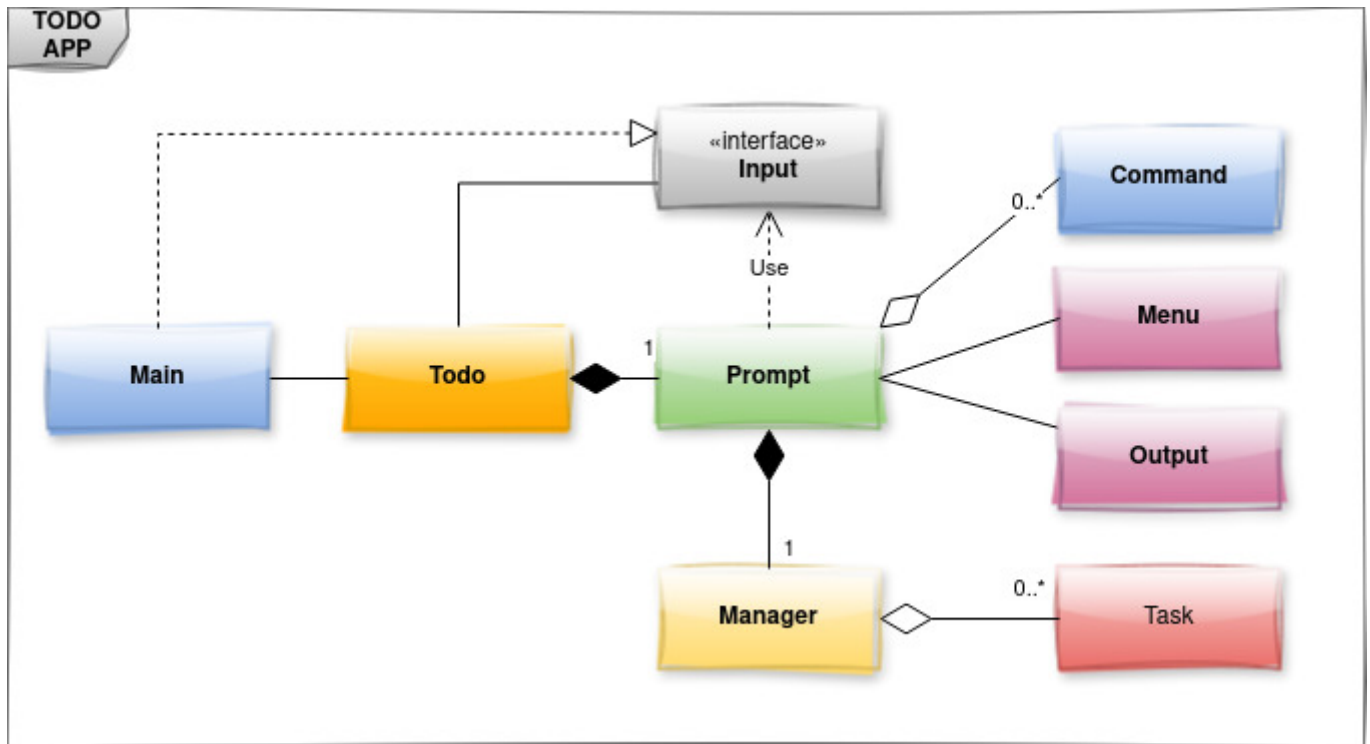
A abstração é o pilar mais difícil de ser compreendido. Este conceito está intimamente relacionado com a modelagem, e modelagem não é uma coisa simples de aprender, demanda tempo e experiência sobre o assunto ou problema que se deseja modelar. É basicamente um trabalho quase que infinito, é preciso constante reavaliação do conhecimento e aplicar quando pertinente, o novo entendimento sobre o modelo conforme o conhecimento se aprofunda no tema. Isto gera refatorações e isso ressalta a importância de ter um código bem escrito.

Para exemplificar a abstração ao invés de modelar uma classe simples, seremos mais arrojados. Vamos tentar modelar uma aplicação bem familiar a todos acredito eu. A famosa lista de tarefas ([Todo List](#)). Com o apoio do [UML](#), modelaremos o relacionamento dos componentes (classes) da aplicação e tentar ver como a idéia de abstrair algo em código funciona.

Lembrando que para este exemplo não vou fazer um comparativo com Java, igual ao que fizemos nos capítulos anteriores, esta atividade fica de exercício para o leitor. Como dito em capítulos anteriores, vou usar a abordagem orientado a erros, que como vocês estão carecas de saber, evita problemas de gerenciamento de memória. E aproveitando e fazendo um jabazinho sobre o meu [Framework](#) conhecido como [SAT \(Swiss Army Tool\)](#) que estou desenvolvendo no canal [Faz em C](#), se não viu ainda, corre lá e se inscreve. Vou utilizar o SAT que me provê uma estrutura de dados para podermos armazenar as tarefas criadas.

Modelando a aplicação

Neste projeto vamos criar uma aplicação [Todo List](#). O diagrama do relacionamento dos componentes é apresentado a seguir:



Para exemplificar, vamos iniciar pelo bloco **Main**, que implementa a interface representada pelo bloco **Input** e tem uma relação de associação com o bloco **Todo**. Aqui até poderia ser uma composição, mas não é factível representarmos o **main** no diagrama, durante a implementação vou explicar o porquê.

Seguindo para o bloco **Todo**. O **Todo** é a aplicação em si, ela é a classe que vai conter toda a aplicação. Aqui é um ponto para ressaltar a importância do leitor conhecer sobre o UML. Se não souber, não tem problema. Vou tentar ser o mais claro possível.

No diagrama é possível notar que o bloco **Todo** tem um relação simples com o bloco **Input**, isto quer dizer que, o **Todo** e o **Input** se relaciona de forma momentânea, que no nosso contexto este relacionamento só existe para que o **Input** chegue até o bloco **Prompt** que de fato usa o **Input**. Ainda no **Todo**, podemos ver que o **Todo** é composto de **Prompt**, isto significa que o **Prompt** só pode existir se o **Todo** existir. Lembrando que isto não é uma regra, é simplesmente a forma que estamos modelando a solução, você poderia modelar de um jeito diferente, em outras palavras, não tem jeito errado.

Seguindo para o bloco **Prompt**. Podemos dizer que aqui é o coração da aplicação, é onde todas as coisas se conectam para dar vida a solução. Olhando para o **Prompt** acima temos o **Input** este relacionamento diz que o **Prompt** precisa do **Input** para funcionar, sem o **Input** o **Prompt** não pode prosseguir e aqui se deve abortar a aplicação. Este padrão é conhecido como injeção de dependência, a letra D do SOLID. Ainda no **Prompt** podemos ver que possui um relacionamento de agregação com o bloco **Command**. Isto significa que o **Prompt** pode ter nenhum ou vários comandos. Podemos ver também que tem um relacionamento simples com o bloco **Menu**. Este relacionamento só serve para apresentar o menu da aplicação, o mesmo acontece com o **Output**. O **Output** é para dar uma animada nos textos e apresentar eles de forma colorida. Abaixo temos o relacionamento de composição com o bloco **Manager**, novamente exprime o relacionamento que **Manager** só existirá se **Prompt** existir, e **Prompt** pode ter somente um **Manager**.

Seguindo para o bloco **Manager**. Aqui temos o último ponto da nossa modelagem. Este relacionamento com o bloco **Task**, diz que **Manager** pode ter 0 ou várias **Tasks**.

Explicado como o diagrama está disposto podemos partir para a implementação propriamente dita.

Vamos iniciar criando o projeto. Aqui eu utilizo `SAT Scaffolding` para criar o projeto base, já com o nome da aplicação e utilizando o `SAT` como `Framework`. Este projeto pode ser visto no canal `Todo`. Lá apresento como instalar criar e instalar. Continuando, abra um terminal e digite:

```
$ sat_scaffolding Todo todo user .
```

O `user`, pode ser o seu nome de usuário. Com o projeto criado, abra com o seu editor favorito. Eu vou de VSCode.

Dentro do projeto podemos encontrar a seguinte organização:

```
Todo
├── build
├── CMakeLists.txt
├── docker-compose.yaml
├── Dockerfile
├── .gitignore
├── include
└── src
```

Esta estrutura inicial já nos permite compilar o projeto sem problemas, facilitando a nossa vida. Daqui por diante é ir incrementando conforme o nosso diagrama. Vamos iniciar a implementação pela parte mais simples o bloco `Task`. Nesta classe vamos aplicar o encapsulamento, que analisa que os argumentos estão devidamente preenchidos para que a `Task` seja criada de forma consistente. A seguir listamos o arquivo `task.h`. Aproveitando para informar que todos os arquivos `header` com a extensão `.h` deverão ser criados na pasta `include`, e para os arquivos fonte com extensão `.c` deverão ser criados na pasta `src`.

```
// task.h

#ifndef TASK_H_
#define TASK_H_

#include <stdbool.h>
#include <stdint.h>

#define TASK_NAME_SIZE          120
#define TASK_DESCRIPTION_SIZE   120

typedef struct
{
    uint32_t id;
    char name [TASK_NAME_SIZE + 1];
    char description [TASK_DESCRIPTION_SIZE + 1];
    bool done;
} task_t;

bool task_create (task_t *object, char *name, char *description);
```

```
bool task_update (task_t *object, task_t *new);
bool task_set_done (task_t *object, bool done);

#endif/* TASK_H_ */
```

No arquivo `task.h` inserimos dois `headers`, o `stdbool.h` que permite fazer uso do `bool` e o `stdint.h`, que permite a definição das variáveis de forma padronizada como o `uint32_t`. Em seguida criamos dois `defines`, `TASK_NAME_SIZE` e `TASK_DESCRIPTION_SIZE` ambos com 120 bytes, que definem o tamanho dos `buffers` para o `name` e para `description`. A seguir definimos a estrutura `task_t`, que contém os seguintes atributos:

- `id` do tipo `uint32_t`, que serve para identificar a atividade.
- `name` do tipo `char` alocado estaticamente com o define `TASK_NAME_SIZE`, que serve para dar nome a atividade.
- `description` do tipo `char` alocado estaticamente com o define `TASK_DESCRIPTION_SIZE`, que serve para descrever a atividade.
- `done` do tipo `bool`, que serve para marcar a atividade como concluída.

Para lidar com este contexto, criamos três funções sendo elas:

- `task_create` - Utilizada para criar a atividade. Esta função recebe dois argumentos: o `name` e o `description`.
- `task_update` - Utilizada para atualizar uma atividade. Esta função recebe uma nova atividade como base de atualização.
- `task_set_done` - Utilizada para marcar se um atividade foi concluída. Esta função recebe um booleano.

Com base nestas definições, vamos implementar as funções no arquivo `task.c`.

```
// task.c
#include <task.h>
#include <string.h>

static bool task_check_args (char *name, char *description)
{
    bool status = false;

    if (name != NULL &&
        strlen (name) > 0 &&
        description != NULL &&
        strlen (description) > 0)
    {
        status = true;
    }

    return status;
}
```

Para verificar se os argumentos estão válidos, faz-se necessário um função auxiliar, para esta finalidade criamos uma função privada: `task_check_args` que recebe os argumentos `name` e `description` e verifica se elas não estão nulas e se possui tamanho, e retorna verdadeiro caso todas condições sejam válidas. Fica de exercício, verificar se algum destes campos não foram preenchidos somente com espaços, que é uma `string` válida. Vamos a implementação da função `task_create`.

```
// task.c
bool task_create (task_t *object, char *name, char *description)
{
    bool status = false;

    if (object != NULL && task_check_args (name, description) == true)
    {
        memset (object, 0, sizeof (task_t));
        static uint32_t id = 0;

        object->id = ++id;
        strncpy (object->name, name, TASK_NAME_SIZE);
        strncpy (object->description, description, TASK_DESCRIPTION_SIZE);
        object->done = false;

        status = true;
    }

    return status;
}
```

Na função `task_create` recebemos dois argumentos o `name` e o `description`, antes de preencher a instância testamos se a instância não é nula e se os argumentos estão válidos com a função `task_check_args` se tudo estiver válido prosseguimos. Realizamos a limpeza da instância e criamos um pseudo identificador, todas as vezes que esta função for chamada, significa que uma nova atividade é criada, então a cada ocorrência incrementamos e copiamos o `name` e o `description` para a instância retornando verdadeiro. A próxima função é `task_update`.

```
// task.c
bool task_update (task_t *object, task_t *new)
{
    bool status = false;

    if (object != NULL && task_check_args (new->name, new->description) == true)
    {
        memset (object->name, 0, TASK_NAME_SIZE);
        memset (object->description, 0, TASK_DESCRIPTION_SIZE);

        strncpy (object->name, new->name, TASK_NAME_SIZE);
        strncpy (object->description, new->description,
TASK_DESCRIPTION_SIZE);
        object->done = false;
    }
}
```

```
        status = true;
    }

    return status;
}
```

A função `task_update` recebe uma nova atividade como argumento para atualizar uma instância já existente. O seu conteúdo é o mesmo que a `task_create`. Um detalhe aqui, é que sempre que uma atividade é atualizada aplicamos a regra de que a sua conclusão deve ser resetada. A próxima é a função `task_set_done`.

```
// task.c
bool task_set_done (task_t *object, bool done)
{
    bool status = false;

    if (object != NULL)
    {
        object->done = done;

        status = true;
    }

    return status;
}
```

A função `task_set_done` recebe um booleano como argumento que alterna o estado de concluída da atividade. Na aplicação utilizamos esta função somente para marcar como concluída, sempre passando um verdadeiro. Com isso concluímos a implementação da `task_t`. Aqui utilizamos o encapsulamento, protegendo as regras de negócio. Fique a vontade para adicionar novas funções e mais verificações. Seguindo para a próxima classe, a classe `Manager`. Para isso, vamos criar o arquivo `task_manager.h`.

```
// task_manager.h
#ifndef TASK_MANAGER_H_
#define TASK_MANAGER_H_

#include <sat.h>
#include <task.h>

#define TASK_MANAGER_TASKS_MAXIMUM 20

typedef struct
{
    sat_array_t *tasks;
} task_manager_t;

bool task_manager_open (task_manager_t *object);
```

```

bool task_manager_add (task_manager_t *object, task_t *task);
bool task_manager_get_by (task_manager_t *object, uint32_t index, task_t
*task);
bool task_manager_find_by (task_manager_t *object, uint32_t id, task_t
*task, uint32_t *index);
bool task_manager_update_by (task_manager_t *object, task_t *task, uint32_t
index);
bool task_manager_remove_by (task_manager_t *object, uint32_t index);
bool task_manager_close (task_manager_t *object);
uint32_t task_manager_get_tasks_amount (task_manager_t *object);

#endif/* TASK_MANAGER_H */

```

No arquivo `task_manager.h` inserimos dois `headers`, `sat.h` e `task.h`. Criamos um `define` para limitarmos o número em 20 atividades que podem ser cadastradas. A seguir definimos a classe `task_manager_t`, que possui uma lista do tipo `array` de `task_t`. Para fazer o gerenciamento desta lista temos as seguintes funções:

- `task_manager_open` - Inicializa a lista de atividades.
- `task_manager_add` - Permite adicionar uma nova atividade.
- `task_manager_get_by` - Recupera uma atividade utilizando um índice.
- `task_manager_find_by` - Recupera uma atividade através do identificador da atividade.
- `task_manager_update_by` - Atualiza uma atividade pelo seu índice.
- `task_manager_remove_by` - Remove uma atividade pelo índice.
- `task_manager_close` - Libera os recursos da lista.
- `task_manager_get_tasks_amount` - Obtém a quantidade de atividades já adicionadas.

Com base nestas definições, vamos implementar as funções no arquivo `task_manager.c`.

```

// task_manager.c

#include <task_manager.h>
#include <string.h>

bool task_manager_open (task_manager_t *object)
{
    bool status = false;

    if (object != NULL)
    {
        sat_status_t __status = sat_array_create (&object->tasks,
TASK_MANAGER_TASKS_MAXIMUM, sizeof (task_t));

        status = sat_status_get_result (&__status);
    }

    return status;
}

```

Nesta primeira implementação, iniciamos adicionando os `headers`, `task_manager.h` e `string.h`. Na função `task_manager_open`, verificamos se o objeto não é nulo, caso não, inicializamos a lista com a quantidade de atividades definida em `TASK_MANAGER_TASKS_MAXIMUM` e com o tamanho do objeto `task_t`. Em caso de sucesso retornamos verdadeiro. Seguindo para a função `task_manager_add`.

```
// task_manager.c
bool task_manager_add (task_manager_t *object, task_t *task)
{
    bool status = false;

    if (object != NULL && task != NULL)
    {
        sat_status_t __status = sat_array_add (object->tasks, (void
*)task);

        status = sat_status_get_result (&__status);
    }

    return status;
}
```

A função `task_manager_add` verifica que a `task` não é nula, caso não adiciona na lista, e retorna verdadeiro. Próxima função `task_manager_get_by`.

```
// task_manager.c
bool task_manager_get_by (task_manager_t *object, uint32_t index, task_t
*task)
{
    bool status = false;

    uint32_t size;

    sat_array_get_size (object->tasks, &size);

    if (object != NULL && task != NULL && index < size)
    {
        memset (task, 0, sizeof (task_t));

        sat_status_t __status = sat_array_get_object_by (object->tasks,
index, (void *)task);

        status = sat_status_get_result (&__status);
    }

    return status;
}
```

A função `task_manager_get_by` recebe um índice e uma `task` como argumento. Primeiro obtemos a quantidade de itens na lista com a função `sat_array_get_size`. No `if` verificamos se a `task` e se o índice é menor que a quantidade de itens na lista. Em caso de tudo está válido, limpamos o objeto, e obtemos o objeto solicitado pelo índice e copiamos para a `task` do usuário, retornando verdadeiro em caso de sucesso. Próxima função `task_manager_find_by`.

```
// task_manager.c
bool task_manager_find_by (task_manager_t *object, uint32_t id, task_t
*task, uint32_t *index)
{
    bool status = false;

    uint32_t size;

    if (object != NULL && task != NULL && index != NULL)
    {
        sat_array_get_size (object->tasks, &size);

        for (uint32_t i = 0; i < size; i++)
        {
            memset (task, 0, sizeof (task_t));
            sat_array_get_object_by (object->tasks, i, task);

            if (task->id == id)
            {
                *index = i;
                status = true;
                break;
            }
        }
    }

    return status;
}
```

A função `task_manager_find_by` recebe um identificador, uma `task` e um índice para retorno. No `if` verificamos se a `task` e o `index` estão válidos. Em caso válido, obtemos a quantidade da lista e iteramos procurando a atividade que corresponde ao identificador informado. Em caso de encontrado, obtemos a posição da atividade na lista e paramos o `loop`, retornando um verdadeiro. A próxima é a função `task_manager_update_by`.

```
// task_manager.c
bool task_manager_update_by (task_manager_t *object, task_t *task, uint32_t
index)
{
    bool status = false;

    if (object != NULL)
    {
```

```
        sat_status_t __status = sat_array_update_by (object->tasks, task,
index);

        status = sat_status_get_result (&__status);
    }

    return status;
}
```

A função `task_manager_update_by` recebe uma `task` e um índice com estes argumentos atualizamos a atividade localizada na posição definida pelo índice, retornando verdadeiro em caso de sucesso. Seguindo para a função `task_manager_remove_by`.

```
// task_manager.c
bool task_manager_remove_by (task_manager_t *object, uint32_t index)
{
    bool status = false;

    if (object != NULL)
    {
        sat_status_t __status = sat_array_remove_by (object->tasks, index);

        status = sat_status_get_result (&__status);
    }

    return status;
}
```

A função `task_manager_remove_by` recebe um índice como argumento e remove uma atividade com a função `sat_array_remove_by` utilizando o índice. Prosseguindo para a função `task_manager_close`.

```
// task_manager.c
bool task_manager_close (task_manager_t *object)
{
    bool status = false;

    if (object != NULL)
    {
        sat_status_t __status = sat_array_destroy (object->tasks);

        status = sat_status_get_result (&__status);
    }

    return status;
}
```

A função `task_manager_close` se encarrega de finalizar o array e liberar os recursos alocados por ela. E finalmente a função `task_manager_get_tasks_amount`.

A função `task_manager_get_tasks_amount` obtém a quantidade de atividades presente na lista e retorna. Isto finaliza a implementação do bloco `Manager`. Novamente nesta implementação praticamos o encapsulamento. Todo o acesso a lista de atividades deve ser solicitado para que o `Manager` o faça, sendo ele o dono da informação. Podemos seguir viagem agora implementando o bloco `Menu`. Aqui é basicamente perfumaria. Beleza realmente importa certo? Ninguém gostaria de comprar uma Ferrari com arranhões na pintura. Então vamos deixar a aplicação apresentável. Vamos criar o arquivo `menu.h`.

O arquivo `menu.h` possui a definição de duas funções somente, a função `menu_logo` e a função `menu_show`. Vamos criar o `menu.c` para implementarmos estas funções.

49 / 70

No arquivo `menu.c` adicionamos os `headers`, `menu.h` e `stdio.h`. Na função `menu_logo` um bânêr é criado com o nome da aplicação. Este tipo de arte é conhecido com `ASCII Art`. Existem gerados online para converter os textos em `ASCII Art`, o que foi utilizado para a criação deste bânêr foi o [Many Tools](#). A implementação da função é somente criar uma variável do tipo ponteiro que retorne o bânêr. Indo para a função `menu_show`.

Na função `menu_show` criamos uma lista de `strings`. Cada `string` representa uma ação que o usuário pode realizar na aplicação. E para apresentar utilizamos um `for` iterando cada texto, assim montando o menu. Agora a aplicação possui uma face. Mas o mundo em preto e branco tem o seu charme, mas não é tão glamuroso como um mundo colorido cheio de possibilidades, não é mesmo? Para atingir este objetivo vamos implementar o bloco `Output` que imprime texto utilizando cores. Vamos criar o arquivo `output.h`.

```
//output.h
#ifndef OUTPUT_H_
#define OUTPUT_H_

typedef enum
{
    output_color_black,
    output_color_red,
    output_color_green,
    output_color_yellow,
    output_color_blue,
    output_color_purple,
    output_color_cyan,
    output_color_white,
} output_color_t;

void output_color_print (const char *text, output_color_t color);
void output_color_print_set (output_color_t color);
void output_color_print_reset ();

#endif/* OUTPUT_H_ */
```

No arquivo `output.h` definimos uma enumeração com as cores que vamos utilizar para imprimir as mensagens. Para abstrair como as cores são utilizadas, criamos três funções sendo elas:

- `output_color_print` - Imprime os textos coloridos.
- `output_color_print_set` - Configura a cor que será utilizada para impressão das mensagens.
- `output_color_print_reset` - Retorna a configuração padrão.

Vamos criar o arquivo `output.c` para implementar as funções descritas:

```
// output.c
#include <output.h>
#include <stdio.h>
#include <stdint.h>

#define OUTPUT_COLOR_RESET        "\033[0m"
#define OUTPUT_COLOR_BLACK       "\033[0;30m"
#define OUTPUT_COLOR_RED         "\033[0;31m"
#define OUTPUT_COLOR_GREEN       "\033[0;32m"
#define OUTPUT_COLOR_YELLOW      "\033[0;33m"
#define OUTPUT_COLOR_BLUE        "\033[0;34m"
#define OUTPUT_COLOR_PURPLE      "\033[0;35m"
#define OUTPUT_COLOR_CYAN        "\033[0;36m"
#define OUTPUT_COLOR_WHITE       "\033[0;37m"
```

No arquivo `output.c` incluímos os `headers`: `output.h`, `stdio.h` e `stdint.h`. Em seguida definimos as cores. Este número estranho é um código de escape ANSI usado para formatar o texto em terminais. Pesquise para saber mais.

```
// output.c
typedef struct
{
    output_color_t color;
    const char *code;
} output_color_pair_t;

static output_color_pair_t pairs [] =
{
    {.color = output_color_black, .code = OUTPUT_COLOR_BLACK},
    {.color = output_color_red, .code = OUTPUT_COLOR_RED},
    {.color = output_color_green, .code = OUTPUT_COLOR_GREEN},
    {.color = output_color_yellow, .code = OUTPUT_COLOR_YELLOW},
    {.color = output_color_blue, .code = OUTPUT_COLOR_BLUE},
    {.color = output_color_purple, .code = OUTPUT_COLOR_PURPLE},
    {.color = output_color_cyan, .code = OUTPUT_COLOR_CYAN},
    {.color = output_color_white, .code = OUTPUT_COLOR_WHITE},
};

static uint8_t output_color_amount = sizeof (pairs) / sizeof (pairs [0]);
```

Para correlacionar a cor que foi definida na enumeração com o seu respectivo código, criamos a estrutura `output_color_pair_t` para mapear enum e código. Utilizando a estrutura criamos um vetor `pairs` mapeando cada uma das cores. Para contar a quantidade de pares que foram formados utilizamos a variável `output_color_amount` que recebe o cálculo do tamanho de `pairs` pela sua unidade `pairs [0]`.

```
// output.c
static char *output_color_by (output_color_t color)
{
    char *code = OUTPUT_COLOR_WHITE;

    for (uint8_t i = 0; i < output_color_amount; i++)
    {
        if (color == pairs [i].color)
        {
            code = pairs [i].code;
            break;
        }
    }

    return code;
}
```

A função `output_color_by` obtém o código em `string` baseado no enum correspondente. O código default inicia com o branco. Em seguida escaneamos o vetor um busca da cor, se encontrar a cor informada, o código é alterado para a cor desejada.

```
// output.c
void output_color_print (const char *text, output_color_t color)
{
    output_color_print_set (color);

    printf ("%s", text);

    output_color_print_reset ();
}
```

Na função `output_color_print` recebemos o texto e a cor desejada para imprimir o texto. A função `output_color_print_set` configura a cor do `output`, logo em seguida o texto é impresso, finalizando a impressão configurando para a cor padrão com a função `output_color_print_reset`.

```
// output.c
void output_color_print_set (output_color_t color)
{
    printf ("%s", output_color_by (color));
}
```

Como já vimos, a função `output_color_print_set` configura o terminal para usar a cor informada.

```
// output.c
void output_color_print_reset ()
{
    printf ("%s", OUTPUT_COLOR_RESET);
}
```

A função `output_color_print_reset` restabelece as configurações `default` do terminal. E finalizamos a implementação do bloco `Output`. O próximo que vamos atacar é o bloco `Input`. Este bloco representa a interface que o usuário precisa preencher com a sua implementação de como vai informar os dados para a aplicação. Aqui enfatizamos o uso poliformismo novamente. Para implementá-la vamos criar o arquivo `input_base.h`.

```
// input_base.h
#ifndef INPUT_BASE_H_
#define INPUT_BASE_H_

#include <stdlib.h>

typedef struct
```

```
{
    void (*read) (char *buffer, size_t size);
} input_base_t;

#endif/* INPUT_BASE_H_ */
```

No arquivo `input_base.h` incluímos o header `stdlib.h` para usar o tipo `size_t` e definimos a interface `input_base_t`, contendo somente um único ponteiro de função: o `read` que recebe um `buffer` e um `size`, que são informados pela aplicação. A função de quem implementa esta interface, é somente preencher o `buffer` utilizando o `size`. Com a interface definida vamos para o coração da aplicação o ponto que unifica todos estes elementos já implementados. Vamos criar o arquivo `prompt.h`.

```
// prompt.h
#ifndef PROMPT_H_
#define PROMPT_H_

#include "input_base.h"
#include <task_manager.h>

#define PROMPT_BUFFER_SIZE 120

typedef struct
{
    input_base_t *base;
    task_manager_t manager;
    char buffer [PROMPT_BUFFER_SIZE + 1];
    bool run;
} prompt_t;

void prompt_open (prompt_t *object, input_base_t *base);
void prompt_run (prompt_t *object);

#endif/* PROMPT_H_ */
```

No arquivo `prompt.h` adicionamos dois headers: `input_base.h` e `task_manager.h`. Criamos um define `PROMPT_BUFFER_SIZE` que define o tamanho do `buffer` que a aplicação vai utilizar. Seguindo criamos a estrutura `prompt_t`, que contém a referência para interface que vai ser injetada pela aplicação, possui uma instância de `task_manager_t`, e um `buffer` com tamanho de `PROMPT_BUFFER_SIZE` e uma variável booleana `run`, que controla a execução da aplicação. Para interagir com este contexto temos duas funções, sendo elas:

- `prompt_open` - Inicia todos os componentes do contexto.
- `prompt_run` - Inicia a execução da aplicação.

Vamos criar o arquivo `prompt.c` para implementar as funções descritas:

```
// prompt.c
#include <prompt.h>
```

```
#include <string.h>
#include <ctype.h>
#include <menu.h>
#include <output.h>

#define PROMPT_ADD_COMMAND      "add"
#define PROMPT_DISPLAY_COMMAND "display"
#define PROMPT_EXIT_COMMAND    "exit"
#define PROMPT_REMOVE_COMMAND  "remove"
#define PROMPT_UPDATE_COMMAND  "update"
#define PROMPT_COMPLETE_COMMAND "complete"
#define PROMPT_SAVE_COMMAND    "save"
```

No arquivo `prompt.c` adicionamos alguns `includes`, o `prompt.h`, `string.h`, `ctype.h`, `menu.h` e `output.h`. Em seguida definimos os comandos que estão descritos no menu da aplicação, já implementado. Quase todos os comandos serão implementados, somente o `save` não será. No final do livro eu explico o porquê.

```
// prompt.c
static void prompt_show (void);
static void prompt_read (prompt_t *object);
static void prompt_read_command (prompt_t *object);
static void prompt_string_to_lower (char *string);
static bool prompt_is_a_number (char *string);
static task_t prompt_fill_task (prompt_t *object);
static bool prompt_ask_for_complete (prompt_t *object, char *text);
static bool prompt_ask_for_id (prompt_t *object, char *text, uint32_t *id);
static bool prompt_are_tasks (prompt_t *object);
static bool prompt_command_process (prompt_t *object);

static void prompt_create_task (prompt_t *object);
static void prompt_display (prompt_t *object);
static void prompt_exit (prompt_t *object);
static void prompt_remove_task (prompt_t *object);
static void prompt_update_task (prompt_t *object);
static void prompt_complete_task (prompt_t *object);
static void prompt_save_task (prompt_t *object);
```

Neste arquivo utilizamos diversas funções de apoio, e para questões de conveniência o bloco `Command` foi absorvido pelo bloco `Prompt`. A relação entre eles existe conforme demonstra o diagrama, porém por questões de simplificação acabei decidindo por deixar no mesmo arquivo. Fica de exercício remover `command` e verificar se o modelo ainda atende. Então vamos as funções.

- `prompt_show` - Apresenta o `prompt` da aplicação para o usuário entrar com um comando.
- `prompt_read` - Lê os dados do teclado.
- `prompt_read_command` - Lê o comando.
- `prompt_string_to_lower` - Altera a `string` para letras minúsculas.

- `prompt_is_a_number` - Verifica se a `string` informada é um número.
- `prompt_fill_task` - Função para preencher uma `task` quando solicitado a sua criação.
- `prompt_ask_for_complete` - Pergunta ao usuário se realmente quer concluir a operação.
- `prompt_ask_for_id` - Obtém um identificador.
- `prompt_are_tasks` - Verifica se tem `tasks` na lista.
- `prompt_command_process` - Processa os comandos.
- `prompt_create_task` - Cria uma nova `task`.
- `prompt_display` - Imprime todas as `tasks` da lista.
- `prompt_exit` - Sai da aplicação.
- `prompt_remove_task` - Remove uma `task`.
- `prompt_update_task` - Atualiza uma `task`.
- `prompt_complete_task` - Marca uma `task` como concluída.
- `prompt_save_task` - Salva a lista.

Como havia dito, são muitas funções, pode ser que o meu modelo não esteja bom o suficiente para você que está lendo. Mas reiterando, não existe modo errado de se modelar. Com o tempo, quando mais se saber sobre o problema, mais claro se torna o entendimento, resultando em remanejamento, remoção ou criação de novos elementos para refinar o modelo.

```
// prompt.c
typedef struct
{
    const char *name;
    void (*command) (prompt_t *object);
} prompt_command_t;

static prompt_command_t commands [] =
{
    {.name = PROMPT_ADD_COMMAND,      .command = prompt_create_task},
    {.name = PROMPT_DISPLAY_COMMAND,  .command = prompt_display},
    {.name = PROMPT_EXIT_COMMAND,     .command = prompt_exit},
    {.name = PROMPT_REMOVE_COMMAND,   .command = prompt_remove_task},
    {.name = PROMPT_UPDATE_COMMAND,   .command = prompt_update_task},
    {.name = PROMPT_COMPLETE_COMMAND, .command = prompt_complete_task},
    {.name = PROMPT_SAVE_COMMAND,     .command = prompt_save_task},
};

static uint8_t prompt_commands_amount = sizeof (commands) / sizeof
(commands [0]);
```

O bloco `Command` está bem aqui. Aqui definimos o `prompt_command_t`, que contém um `name`, sendo este o nome do comando, que está associando o comando com a sua ação representado pelo ponteiro de função. Em seguida realizamos o mapeamento com o vetor `commands` entre os comandos e as suas respectivas ações. E para ter a quantidade de comandos, criamos uma variável `prompt_commands_amount` que contém o resultado do cálculo que obtém esta quantidade. Para para a verificação da primeira função implementada, o `prompt_open`.


```
// prompt.c
void prompt_open (prompt_t *object, input_base_t *base)
{
    object->base = base;
    object->run = true;

    task_manager_open (&object->manager);
}
```

A função `prompt_open` obtém a interface `input_base_t` via injeção de dependência atribuindo ao seu contexto, e já setando para `true` o `run`. Em seguida a função `task_manager_open` é invocada para iniciar o `manager`. A próxima é o `prompt_run`.

```
// prompt.c
void prompt_run (prompt_t *object)
{
    while (object->run)
    {
        output_color_print (menu_logo (), output_color_cyan);
        menu_show ();
        prompt_show ();

        prompt_read_command (object);

        if (prompt_command_process (object) == false)
        {
            output_color_print ("Invalid command.\n\n", output_color_red);
        }
    }
}
```

A função `prompt_run` coloca a aplicação em execução. Dentro de um loop `while`, verifica se a aplicação está em execução, apresenta o logo com a função `output_color_print` na cor ciano. Apresenta o menu de opções com o `menu_show` e o prompt com o `prompt_show` para o usuário inserir o comando e aguarda uma ação por parte do usuário com o `prompt_read_command`.

Uma vez que o comando foi inserido, podemos processá-lo com o `prompt_command_process`, caso retorne erro apresentamos que o comando é inválido na cor vermelha.

```
// prompt.c
static void prompt_show (void)
{
    output_color_print ("(todo) > ", output_color_cyan);
}
```

A função `prompt_show` imprime a string `(todo) >` no terminal na cor ciano.

```
// prompt.c
static void prompt_read (prompt_t *object)
{
    memset (object->buffer, 0, PROMPT_BUFFER_SIZE);

    object->base->read (object->buffer, PROMPT_BUFFER_SIZE);
}
```

A função `prompt_read` limpa o `buffer` e solicita um comando através da interface utilizando o `buffer` e o `PROMPT_BUFFER_SIZE`.

```
// prompt.c
static void prompt_read_command (prompt_t *object)
{
    prompt_read (object);
    prompt_string_to_lower (object->buffer);
}
```

A função `prompt_read_command` existe para facilitar a nossa vida. Ela obtém a leitura quando se trata de um comando, e muda todos os caracteres para minúsculo para evitar qualquer problema devido ao `case sensitive`.

```
// prompt.c
static void prompt_string_to_lower (char *string)
{
    for (int i = 0; i < strlen (string); i++)
    {
        string [i] = tolower (string [i]);
    }
}
```

A função `prompt_string_to_lower` coloca todos os caracteres para minúsculos com a função `tolower`.

```
// prompt.c
static void prompt_create_task (prompt_t *object)
{
    char *text = "You are about to add a new task. Are you sure? (yes/no)";

    task_t task = prompt_fill_task (object);

    if (prompt_ask_for_complete (object, text) == true)
    {
        task_manager_add (&object->manager, &task);
        output_color_print ("New task added successully.\n\n",
            output_color_green);
    }
}
```

```

    }
}

```

Na função `prompt_create_task` a `task` é criada com a função `prompt_fill_task`, logo em seguida, uma pergunta é feita com a função `prompt_ask_for_complete` para o usuário se ele quer realmente criar essa nova atividade. Sendo a resposta positiva, a `task` é passada para o `manager` adicionar a `task` a lista com a função `task_manager_add` apresentando uma mensagem de sucesso.

```

// prompt.c
static void prompt_display (prompt_t *object)
{
    task_t task;

    for (uint8_t i = 0; i < TASK_MANAGER_TASKS_MAXIMUM; i++)
    {
        if (task_manager_get_by (&object->manager, i, &task) == true)
        {
            printf ("%d. [%c] - %s - %s\n", task.id, task.done == true ?
'X' : ' ', task.name, task.description);
        }
    }

    printf ("\n\n");
}

```

A função `prompt_display` escaneia a lista obtendo todos os registros e imprimindo cada um deles.

```

// prompt.c
static void prompt_remove_task (prompt_t *object)
{
    char *text = "Would you like to remove? (yes/no): ";
    char *text_id = "Type the task id to delete or exit to cancel: ";
    uint32_t id;

    if (prompt_are_tasks (object) == true)
    {
        if (prompt_ask_for_id (object, text_id, &id) == true)
        {
            task_t task;
            uint32_t index;

            if (task_manager_find_by (&object->manager, id, &task, &index)
== true)
            {
                printf ("Found: %d. [%c] - %s - %s\n", task.id, task.done
== true ? 'X' : ' ',
                                                                    task.name,
                                                                    task.description);
            }
        }
    }
}

```

```

        if (prompt_ask_for_complete (object, text) == true)
        {
            task_manager_remove_by (&object->manager, index);
            output_color_print ("Task removed successully.\n\n",
output_color_green);
        }
    }
}
}
}

```

A função `prompt_remove_task` verifica se há `tasks` na lista, em caso de houver alguma, a aplicação pede que o usuário informe um `id`, se o `id` for encontrado pela função `task_manager_find_by`, a mensagem dos dados da `task` encontrada é apresentada. A aplicação pergunta se o usuário quer seguir com a deleção. Se o usuário confirmar, a `task` será removida pela função `task_manager_remove_by` apresentando uma mensagem de sucesso.

```

// prompt.c
static void prompt_update_task (prompt_t *object)
{
    char *text = "Would you like to update? (yes/no): ";
    char *text_id = "Type the task id to update or exit to cancel: ";

    uint32_t id;

    if (prompt_are_tasks (object) == true)
    {
        if (prompt_ask_for_id (object, text_id, &id) == true)
        {
            task_t task;
            uint32_t index;

            if (task_manager_find_by (&object->manager, id, &task, &index)
== true)
            {
                printf ("Found: %d. [%c] - %s - %s\n", task.id, task.done
== true ? 'X' : ' ', task.name, task.description);

                if (prompt_ask_for_complete (object, text) == true)
                {
                    task_t __task = prompt_fill_task (object);

                    task_update (&task, &__task);

                    task_manager_update_by (&object->manager, &__task,
index);

                    output_color_print ("Task update successully.\n\n",
output_color_green);
                }
            }
        }
    }
}

```

```

    }
}
}

```

A função `prompt_update_task` verifica se há `tasks` na lista antes de prosseguir. Caso haja, um `id` é solicitado para o usuário, sendo o `id` válido a `task` é procurada na lista baseado no `id`. Se encontrado apresenta as informações da `task` perguntando se o usuário quer prosseguir com a atualização. Caso seja uma resposta positiva. A função `prompt_fill_task` é chamada para que o usuário atualize os dados. Uma vez preenchida a `task` original é atualizada com a função `task_update`, e consequentemente atualizada na lista com a função `task_manager_update_by` apresentando uma mensagem de sucesso.

```

// prompt.c
static void prompt_complete_task (prompt_t *object)
{
    char *text = "Would you like to mark as complete? (yes/no): ";
    char *text_id = "Type the task id to mark as complete or exit to
cancel: ";

    uint32_t id;

    if (prompt_are_tasks (object) == true)
    {
        if (prompt_ask_for_id (object, text_id, &id) == true)
        {
            task_t task;
            uint32_t index;

            if (task_manager_find_by (&object->manager, id, &task, &index)
== true)
            {
                printf ("Found: %d. [%c] - %s - %s\n", task.id, task.done
== true ? 'X' : ' ', task.name, task.description);

                if (prompt_ask_for_complete (object, text) == true)
                {
                    task_set_done (&task, true);

                    task_manager_update_by (&object->manager, &task,
index);

                    output_color_print ("Task update successully.\n\n",
output_color_green);
                }
            }
        }
    }
}

```

A função `prompt_update_task` verifica se há `tasks` na lista antes de prosseguir. Caso haja, um `id` é solicitado para o usuário, sendo o `id` válido a `task` é procurada na lista baseado no `id`. Se encontrado apresenta as informações da `task` perguntando se o usuário quer prosseguir com a ação. Caso seja uma resposta positiva. A função `task_set_done` é chamada para marcar a `task` como concluída e consequentemente sendo atualizada na lista com a função `task_manager_update_by` apresentando uma mensagem de sucesso.

```
// prompt.c
static void prompt_save_task (prompt_t *object)
{
    (void) object;

    printf ("Save\n\n");
}
```

A função `prompt_save_task` não faz nada, deixando somente o esqueleto para que o leitor a preencha.

```
// prompt.c
static void prompt_exit (prompt_t *object)
{
    object->run = false;
}
```

A função `prompt_exit` altera o estado da variável `run` para encerrar a aplicação.

```
// prompt.c
static bool prompt_is_a_number (char *string)
{
    size_t length = strlen (string);
    bool status = true;

    for (size_t i = 0; i < length; i++)
    {
        if (isdigit (string [i]) == 0)
        {
            status = false;
            break;
        }
    }

    return status;
}
```

A função `prompt_is_a_number` recebe uma `string` e verifica se é um número.

```
// prompt.c
static task_t prompt_fill_task (prompt_t *object)
{
    task_t task ;

    char name [TASK_NAME_SIZE + 1] = {0};
    char description [TASK_DESCRIPTION_SIZE + 1] = {0};

    fprintf (stdout, "Type the task name: ");
    object->base->read (name, TASK_NAME_SIZE);

    fprintf (stdout, "Type the task description: ");
    object->base->read (description, TASK_DESCRIPTION_SIZE);

    task_create (&task, name, description);

    return task;
}
```

A função `prompt_fill_task` obtém os dados da `task` através da interface, e cria uma `task` com a função `task_create` com o dados informados.

```
// prompt.c
static bool prompt_ask_for_complete (prompt_t *object, char *text)
{
    bool status = false;

    while (true)
    {
        fprintf (stdout, "%s", text);

        char buffer [11] = {0};
        object->base->read (buffer, 10);

        if (strncmp (buffer, "yes", strlen ("yes")) == 0)
        {
            status = true;
            break;
        }

        else if (strncmp (buffer, "no", strlen ("no")) == 0)
        {
            output_color_print ("Canceled.\n\n", output_color_red);
            break;
        }

        else
        {
            output_color_print ("Invalid option.\n\n", output_color_red);
            continue;
        }
    }
}
```

```
    }

    return status;
}
```

A função `prompt_ask_for_complete` fica confinada no `loop while` apresentando a mensagem recebida via argumento e aguardando uma ação do usuário informando as possíveis ações que podem ser realizadas neste contexto, que pode ser um `yes` ou `no`. Se o usuário colocar qualquer outra coisa a mensagem de `Invalid option` será apresentada.

```
// prompt.c
static bool prompt_ask_for_id (prompt_t *object, char *text, uint32_t *id)
{
    bool status = false;

    while (true)
    {
        char buffer [11] = {0};
        printf ("%s", text);
        object->base->read (buffer, 10);

        if (strcmp (buffer, "exit") == 0)
        {
            break;
        }

        else if (prompt_is_a_number (buffer) == true)
        {
            *id = atoi (buffer);
            status = true;
            break;
        }

        else
        {
            printf ("Type a valid ID number\n\n");
        }
    }

    return status;
}
```

A função `prompt_ask_for_id` pede por um `id`, ou se o usuário quiser cancelar, ele pode digitar `exit` para sair. Se um `id` for informado a função `prompt_is_a_number` vai verificar. Se for inválido, como por exemplo `1s0`, isto caracteriza um `id` inválido e a mensagem `Type a valid ID number` aparecerá. Se o `id` for válido a conversão será realizada pela função `atoi`, devolvendo o `id` para a aplicação.


```
// prompt.c
static bool prompt_are_tasks (prompt_t *object)
{
    bool status = true;

    uint32_t amount = task_manager_get_tasks_amount (&object->manager);

    if (amount == 0)
    {
        output_color_print ("There is no task to remove. Backing to main
menu.\n\n", output_color_red);
        status = false;
    }

    return status;
}
```

A função `prompt_are_tasks` verifica se tem alguma `task` na lista, a quantidade é obtido com a função `task_manager_get_tasks_amount`. Se não houver uma mensagem `There is no task to remove. Backing to main menu` será apresentada.

```
// prompt.c
static bool prompt_command_process (prompt_t *object)
{
    bool status = false;

    for (uint8_t i = 0; i < prompt_commands_amount; i++)
    {
        if (strncmp (object->buffer, commands [i].name , strlen (commands
[i].name)) == 0)
        {
            commands [i].command (object);

            status = true;
            break;
        }
    }

    return status;
}
```

A função `prompt_command_process` verifica o comando digitado comparando com a tabela de comandos. Se o comando for encontrado a função correspondente será executada. A seguir vamos somente pontuar os pontos de alteração. O código foi auto gerado pela ferramenta `SAT Scaffolding`.

```
// application.h
#ifndef APPLICATION_H_
```

```
#define APPLICATION_H_

#include <sat.h>
#include <input_base.h>
#include <prompt.h>

typedef struct
{
    prompt_t prompt;
} todo_t;

typedef struct
{
    input_base_t base;
} todo_args_t;

sat_status_t todo_init (todo_t *object);
sat_status_t todo_open (todo_t *object, todo_args_t *args);
sat_status_t todo_run (todo_t *object);
sat_status_t todo_close (todo_t *object);

#endif/* APPLICATION_H_ */
```

No header `application.h` adicionamos os headers `input_base.h` e o `prompt.h`. No contexto da aplicação `todo_t` adicionamos os `prompt_t`. No `todo_args_t` adicionamos a interface `input_base_t`. Conforme o diagrama, ela vai ser repassada para o `prompt_t`.

```
// application.c
#include <application.h>
#include <string.h>

sat_status_t todo_init (todo_t *object)
{
    sat_status_t status = sat_status_set (&status, false, "todo init error");

    if (object != NULL)
    {
        memset (object, 0, sizeof (todo_t));

        sat_status_set (&status, true, "");
    }

    return status;
}
```

A função `todo_init` limpa o contexto da aplicação.

```
// application.c
sat_status_t todo_open (todo_t *object, todo_args_t *args)
{
    sat_status_t status = sat_status_set (&status, false, "todo open
error");

    if (object != NULL && args != NULL && &args->base != NULL)
    {
        prompt_open (&object->prompt, &args->base);
        sat_status_set (&status, true, "");
    }

    return status;
}
```

A função `todo_open` recebe o `todo_args_t` via argumentos e repassa a interface para o `prompt_open`.

```
// application.c
sat_status_t todo_run (todo_t *object)
{
    sat_status_t status = sat_status_set (&status, false, "todo run
error");

    if (object != NULL)
    {
        prompt_run (&object->prompt);

        sat_status_set (&status, true, "");
    }

    return status;
}
```

A função `todo_run` chama o `prompt_run` para que a aplicação seja iniciada.

```
// application.c
sat_status_t todo_close (todo_t *object)
{
    sat_status_t status = sat_status_set (&status, false, "todo close
error");

    if (object != NULL)
    {
        sat_status_set (&status, true, "");
    }

    return status;
}
```

A função `todo_close` encerra aplicação chamando todos os `closes`. Aqui deixei para o leitor implementar. Agora vamos para o último bloco `Main` que é a última parte da aplicação o arquivo `main.c`.

```
// main.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <application.h>

static void main_read (char *buffer, size_t size)
{
    fgets (buffer, size, stdin);

    buffer [strlen (buffer) - 1] = 0;
}
```

No arquivo `main.c` implementamos a interface com o nome de `main_read` conforme o diagrama. Poderíamos ter colocado ela em outro lugar, como por exemplo em um novo arquivo, mas por questões de simplificação utilizamos o arquivo `main` para isso.

```
// main.c
int main(int argc, char *argv[])
{
    sat_status_t status;

    todo_t todo;
    todo_args_t args =
    {
        .base.read = main_read
    };

    do
    {
        status = todo_init (&todo);
        if (sat_status_get_result (&status) == false)
        {
            break;
        }

        status = todo_open (&todo, &args);
        if (sat_status_get_result (&status) == false)
        {
            break;
        }

        status = todo_run (&todo);
        if (sat_status_get_result (&status) == false)
```

```

        {
            break;
        }

        else
            status = todo_close (&todo);

    } while (false);

    printf ("%s\n", sat_status_get_motive (&status));

    return sat_status_get_result (&status) == true ? EXIT_SUCCESS :
EXIT_FAILURE;
}

```

A implementação do `main` é a implementação padrão gerado pelo `SAT Scaffolding`, salvo a variável `args`, que alteramos para receber a referência da interface.

Isto conclui a implementação. Para finalizar basta adicionar todos os arquivos `.c` ao `CMakeLists.txt` para compilar e testar a aplicação. O resultado da aplicação fica com esta cara:

```

#####: '#####: '#####: '#####: '#####: '#####: '#####: '#####:
... ##: '##: ##: ##: '##: ##: ##: ##: '##: '##: '##: '##: '##:
:: ##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##:
:: ##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##:
:: ##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##:
:: ##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##:
:: ##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##: '##:
:: ##: '#####: '#####: '#####: '#####: '#####: '#####: '#####:
:.....:.....:.....:.....:.....:.....:.....:.....:
Add      To add a new item
Remove   To remove a item
Update   To update a item
Display  To display items
Complete To mark item as complete
Save     To save the modifications
Exit     To quit application

(todo) > 

```

Agora é só testar a aplicação. Existem diversos pontos que não implementamos por questões de simplificação, como tratamento de erros em vários pontos, fica de exercício para o leitor.

Mas com isto concluímos a nossa abstração. Para o nosso exemplo abstraímos uma aplicação. Mas a abstração pode ter várias granularidade. Se olharmos atentamente abstraímos a idéia de atividade dentro de uma `task`. Isto demonstra que a abstração atinge diversas camadas de abstração.

Conclusão

Neste capítulo abordamos a Abstração, o pilar mais importante do paradigma orientado a objetos. E para ilustrá-la desenvolvemos um projeto aplicando os conceitos e modelagem para desenvolver a aplicação `Todo List`. O projeto ainda pode ser melhorado trazendo mais recursos como, salvar as atividades em

arquivos ou banco de dados utilizando o conceito de polimorfismo. Ao longo do livro vimos como implementar todos os pilares do paradigma orientado a objetos em Linguagem C. De todos os pilares o único que não recomendo é a herança, o restante podemos utilizar sem problemas. Espero que tenha gostado do livro e espere por mais.

Antes de ir. Não se esqueça de ir no meu canal do YouTube [Faz em C](#), para conferir os meus vídeos onde apresento o desenvolvimento de muitos outros projetos e recursos sobre a linguagem C, e como eu estou construindo o meu Framework.

Confira a versão em vídeo no canal [Faz em C](#). Aproveita e já dá aquela moral. Se inscreva e dá um like. Vamos difundir a informação.