

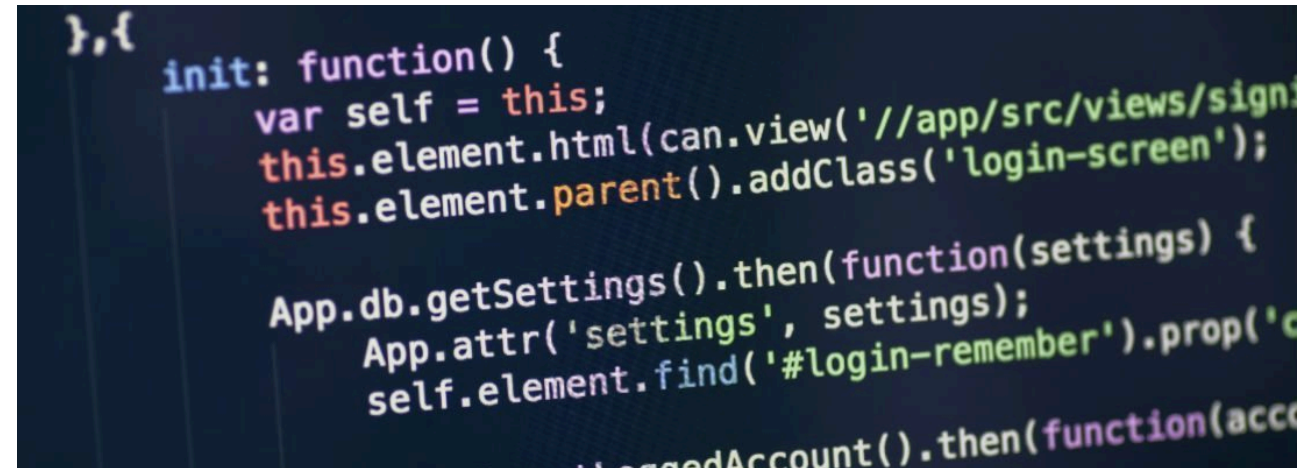
TÓPICO 06 - ESTRUTURA E FORMATAÇÃO

Clean Code - Professor Ramon Venson - SATC 2025

Objetivo da Formatação

A formatação do código esclarece a comunicação do código.

A formatação sobrevive mesmo quando a funcionalidade escrita já não existe mais.



```
}, {  
  init: function() {  
    var self = this;  
    this.element.html(can.view('//app/src/views/signin.html', {  
      this.element.parent().addClass('login-screen');  
    }));  
    App.db.getSettings().then(function(settings) {  
      App.attr('settings', settings);  
      self.element.find('#login-remember').prop('checked', settings.loginRemember);  
    });  
    self.loggedAccount().then(function(account) {  
      self.element.find('#login-remember').prop('checked', account.logged);  
    });  
  }  
});
```

```
64 private int selectedArea;
65 private long areaId;
66 private String areaText;
67
68 private String contactEmail;
69 private String contactPhone;
70 private String contactWeb;
71 private String contactNotes;
72
73 private String debtorOrgNo;
74 private String debtorName;
75 private String debtorAddress;
76 private String debtorZip;
77 private String debtorCity;
78
79 private Long duplicateOfTicketId;
80
81 private java.sql.Date dueDate;
82 private java.sql.Date remindDate;
83 private java.sql.Date dunningDate;
84 private java.sql.Date legalActionDate;
85 private java.sql.Date payDate;
86 private java.sql.Date cancelDate;
87 private String cancelReason;
88
```

Formatação Vertical

Arquivos com menos linhas são inevitavelmente mais simples de ler.

Ter que rolar a tela as vezes é inevitável, mas impede que o leitor tenham uma visão global do código.

Metáfora do Jornal

Um jornal é lido verticalmente. No topo está a manchete, apresentando a matéria. O primeiro parágrafo apresenta uma sinopse do conteúdo.

A história é contada de forma progressiva, com o leitor recebendo mais detalhes conforme avança.



Espaçamento Vertical

Linhas em branco separando diferentes conceitos do código são importantes para diferenciar sua estrutura.

É comum que a declaração de funções/métodos sejam separadas por uma (ou mais) linha em branco.

```

60 $ = await axInstance.get(options.uriCatalonia)
61 let pageXofY = $('div.relative.flex-auto.f6.lh-copy.gray.tr').text()
62 let usersSoFar = /- (\d+)/.exec(pageXofY)[1]
63 let totalUsers = parseInt(/of (\d+,\d+)/.exec(pageXofY)[1].replace(',', ''))
64 $('>div>div>div').filter('.flex').each(function(i, elem) {
65
66     let etiquetaNombre = elem.childNodes[1].children[0].innerText
67     let [, edad, sexo, rol] = /(\d+)(\.) (\.)/.exec(etiquetaNombre)
68     let fotoURL = elem.children[0].children[0].children[0].currentSrc
69     let perfilURL = elem.children[1].children[0].children[0].href
70     let userID = /users\/(\d+)/.exec(perfilURL)[1]
71     let nombre = elem.children[1].children[0].children[0].innerText
72
73     process.stdout.write('✓ \n')
74
75     process.stdout.write('Saving data on page 1... ')
76     usersdb.set(userID, {nombre, perfilURL, fotoURL, edad, sexo, rol})
77     process.stdout.write('✓ \n')

```

Ao invés de:

```
import db from 'db';
const baseUrl = 'https://api.example.com';
function getUser(id) {
  const user = db.get(id);

  return user;
}
function setUser(user) {
  db.set(user);
}
```

Exemplo:

```
import db from 'db';

const baseUrl = 'https://api.example.com';

function getUser(id) {
  const user = db.get(id);
  return user;
}

function setUser(user) {
  db.set(user);
}
```

Utilize as linhas em branco para separar diferentes conceitos.

Continuidade Vertical

Se diferentes conceitos devem ser separados, conceitos similares, que estão relacionados, devem ser mantidos juntos.

Adicionar espaços arbitrários entre as linhas também dificulta a leitura.

```
(function () {  
    'use strict';  
  
    angular  
        .module('decouple.cart')  
        .factory('cartHttpService', cartHttpService);  
  
    function cartHttpService($resource) {  
        var cartResource = $resource('/api/cart'),  
            cartUserResource = $resource('/api/cart/users'),  
            cartItemResource = $resource('/api/cart/items'),  
            cartItemResourceCreate = $resource('/api/cart/items', {}, {  
                save: {  
                    method: 'POST'  
                }  
            }),  
            cartItemResourceUpdate = $resource('/api/cart/items/{ciId}', {}, {  
                update: {  
                    method: 'PUT'  
                }  
            }),  
            cartItemResourceDelete = $resource('/api/cart/items', {}, {  
                delete: {  
                    method: 'DELETE'  
                }  
            });  
  
        return {  
            getCart                : getCart,  
            getCartItem            : getCartItem,  
            getCartItems           : getCartItems,  
            createCartItem         : createCartItem,  
            deleteCartItem         : deleteCartItem,  
            updateCartItem         : updateCartItem,  
            // getArchiveCartItems : getArchiveCartItems  
        };  
    }  
  
    function getCart(ldap, deptId, classId, fullCart) {  
        return cartResource.get({  
            createdBy: ldap,  
            deptId: deptId,  
            classId: classId,  
            fullCart: fullCart  
        }).$promise;  
    }  
})
```


Ao invés de:

```
public project(axis: Vector): Projection {  
    const scalars = [];  
  
    const point = this.center;  
  
    const dotProduct = point.dot(axis);  
    scalars.push(dotProduct);  
  
    scalars.push(dotProduct + this.radius);  
  
    scalars.push(dotProduct - this.radius);  
    return new Projection(Math.min.apply(Math, scalars), Math.max.apply(Math, scalars));  
}
```

Ao use de:

```
public project(axis: Vector): Projection {  
  const scalars = [];  
  const point = this.center;  
  const dotProduct = point.dot(axis);  
  scalars.push(dotProduct);  
  scalars.push(dotProduct + this.radius);  
  scalars.push(dotProduct - this.radius);  
  return new Projection(Math.min.apply(Math, scalars), Math.max.apply(Math, scalars));  
}
```



CATCH INFINITE SCROLL

Distância Vertical

A distância vertical entre a declaração de uma função e sua chamada deve ser a mínima possível.

O mesmo acontece para outras estruturas de código.

Declaração de Variáveis

Variáveis devem ser declaradas próximas às suas primeiras utilizações.

Casos onde a variável será usada durante todo o escopo são exceções a essa regra.

Ao invés de:

```
function createMainElement(tag) {  
  const element = document.createElement(tag);  
  const mainElement = document.querySelector('.main-element');  
  element.classList.add('new-class');  
  mainElement.appendChild(element);  
  return mainElement;  
}
```

Use:

```
function createMainElement(tag) {  
  const element = document.createElement(tag);  
  element.classList.add('new-class');  
  const mainElement = document.querySelector('.main-element');  
  mainElement.appendChild(element);  
  return mainElement;  
}
```

A declaração das variáveis foi reorganizada de forma que elas fiquem próximas às suas primeiras utilizações.

Declaração de Atributos

Em orientação a objetos, atributos são geralmente declarados logo abaixo da assinatura da classe. Em C++, isso é geralmente feito no final da classe.

O importante é que o programador saiba onde encontrá-los.

Ao invés de:

```
class User {  
  nome: string;  
  
  updateNome(nome: string) {  
    this.nome = nome;  
  }  
  
  idade: number;  
  
  updateIdade(idade: number) {  
    this.idade = idade;  
  }  
  
  altura: number;  
  
  updateAltura(altura: number) {  
    this.altura = altura;  
  }  
}
```


Use:

```
class User {  
  nome: string;  
  idade: number;  
  altura: number;  
  
  updateNome(nome: string) {  
    this.nome = nome;  
  }  
  
  updateIdade(idade: number) {  
    this.idade = idade;  
  }  
  
  updateAltura(altura: number) {  
    this.altura = altura;  
  }  
}
```

Funções Dependentes

Sempre que chamamos uma função, sua declaração deve estar próximo à chamada.

Exceções são feitas obviamente à chamada de métodos de um objeto.

Ao invés de:

```
function makeRequest(url) {  
    /* ... */  
    const response = fetch(url);  
    /* ... */  
}  
  
function resolveUrl(url) {  
    /* ... */  
}  
  
function fetch(url) {  
    /* ... */  
    resolveUrl(url);  
    /* ... */  
}
```

Use:

```
function makeRequest(url) {  
  /* ... */  
  const response = fetch(url);  
  /* ... */  
}  
  
function fetch(url) {  
  /* ... */  
  resolveUrl(url);  
  /* ... */  
}  
  
function resolveUrl(url) {  
  /* ... */  
}
```

As funções foram ordenadas na ordem em que são chamadas.

Afinidade Conceitual

Variáveis ou atributos com conceitos relacionados devem ser mantidos juntos. O mesmo acontece para um conjunto de funções ou métodos com similaridades.

Alguns projetos costumam utilizar também ordem alfabética para facilitar a localização de conceitos.

Ao invés de:

```
const userService = new UserService(userRepository);  
const petService = new PetService(petRepository);  
const mapaRepository = new MapRepository();  
const userRepository = new UserRepository();  
const userController = new UserController(userService);  
const petRepository = new PetRepository();  
const mapaService = new MapService(mapaRepository);  
const mapaController = new MapController(mapaService);  
const petController = new PetController(petService);
```

Use:

```
const userController = new UserController(userService);  
const userService = new UserService(userRepository);  
const userRepository = new UserRepository();  
  
const mapaController = new MapController(mapaService);  
const mapaService = new MapService(mapaRepository);  
const mapaRepository = new MapRepository();  
  
const petController = new PetController(petService);  
const petService = new PetService(petRepository);  
const petRepository = new PetRepository();
```

Ordenação Vertical

Em códigos *top-down*, funções devem ser declaradas logo abaixo de onde são chamadas.

O contrário acontece para códigos *bottom-up*.

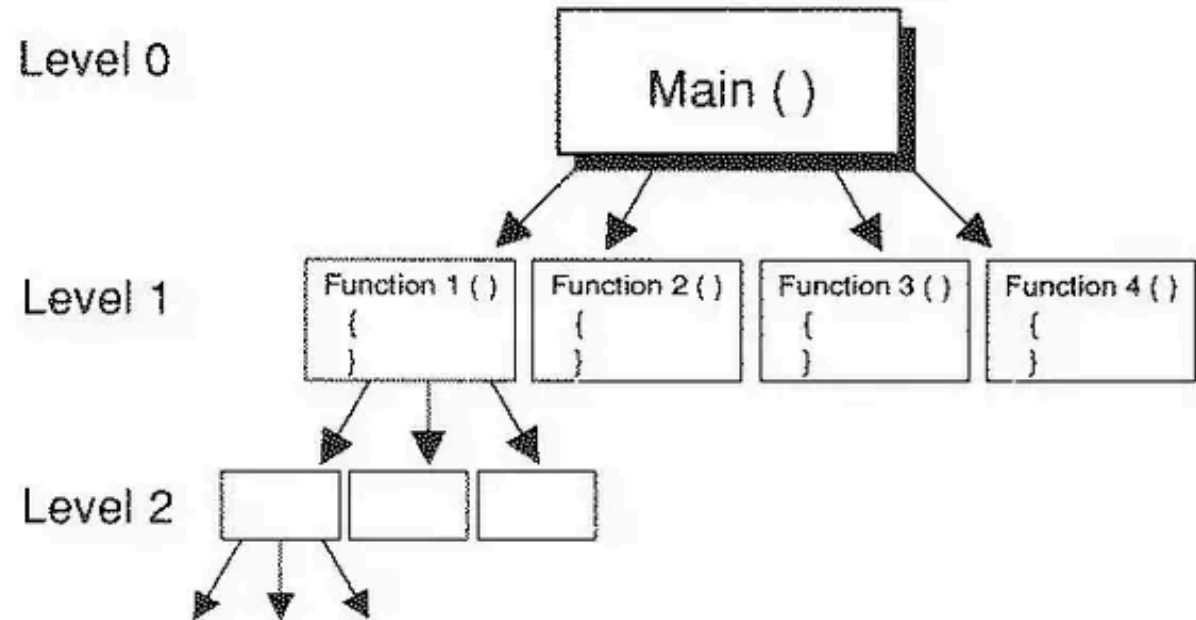


Figure 7.1. C Programs are built from functions.

Formatação Horizontal

Como regra geral, é importante que uma linha faça apenas uma coisa.

No entanto, nomes de funções e variáveis podem ser longos a ponto de que seja necessário rolar a tela para ler o código.

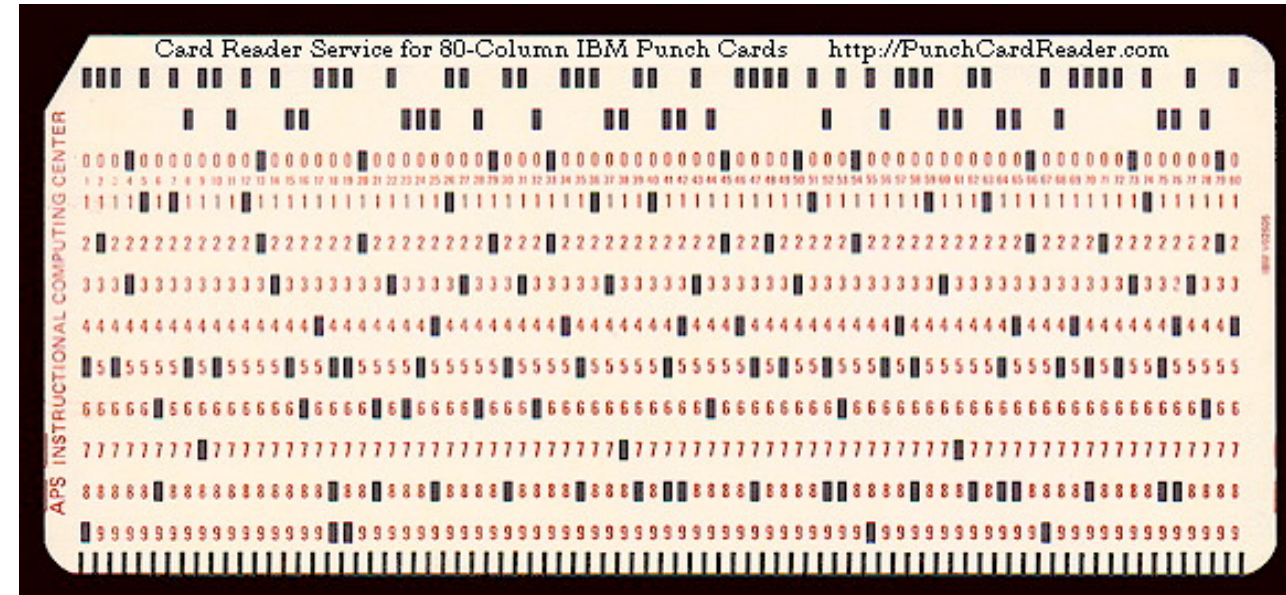
```
/*  
* This is a single block which overflows the default maximum line-length (80 characters).  
* Since this line has a different indentation level it will be considered as a separate block (which also overflows!)  
*  
* @property {"directive"}[] [fixType] An example highlighting that @JSDoc comments will be ignored.  
*  
* @example  
* ```tsx  
* const someValueAfterProcessing = process(value, (node) => ({  
*   ...node,  
*   newProp: 2, // @TODO, insert the correct value of newProp once available here. Do note that I overflow,  
*               // but do not trigger an automatic fix, as code blocks in comments are ignored.  
* }));  
* ```  
*/
```

Limite de Linhas

Os cartões perfurados da IBM eram limitados a 80 caracteres.

Isso estabeleceu uma convenção de que linhas de código não devem ultrapassar 80 caracteres.

Por muito tempo esse padrão também era reforçado pelos monitores em formato 4:3,



```
ItemBuilder: (context, index2) {  
    var tagItem = item  
        .like?.entries  
        .toList()[index2];  
    switch (tagItem?.key ?? "") {  
        case "favorite":  
            return Container(  
                decoration:  
                    BoxDecoration(  
                        color: Colors  
                            .grey[200],  
                        borderRadius:  
                            BorderRadius  
                                .circular(  
                                    16), // BorderRadius.  
                    ), // BoxDecoration
```

Média de Caracteres

Para estabelecer um padrão, pode-se utilizar as recomendações da maioria dos padrões de editores de texto e linguagens:

No máximo, de 80 a 120 caracteres por linha.

Espaçamento e Continuidade Horizontal

Utilize espaço horizontal pra separar elementos que não são intimamente ligados.

Não utilize espaço, por exemplo, para separar os parênteses de uma chamada de função.

```
8   let string = "lorem ipsum";
9   string = "bacon";
10
11  function myFunc ( abc ) {
12    const a = abc;
13
14    const wrongIndent = {
15      a: 'this should have a semi colom'
16    }
17  }
18
```

Ao invés de:

```
function soma (a, b){  
  return a+b ;  
}  
  
soma (1 , 2);
```

Use:

```
function soma(a, b) {  
    return a + b;  
}  
  
soma(1, 2);
```

- Foi removido espaçamentos desnecessários entre nome de função e parênteses.
- Foi adicionado espaço antes e após o operador matemático `+`
- Foi adicionado espaço antes da chave de abertura da função `{`.

Alinhamento Horizontal

Alguns programadores preferem alinhar elementos de código de maneira horizontal.

A declaração de atributos é um exemplo de código que pode ser alinhado.

```
typedef struct
{
    vec3_t    mins, maxs;
    vec3_t    origin;    // for sounds or lights
    float     radius;
    int       headnode;
    int       visleafs;    // not including the solid leaf 0
    int       firstface, numfaces;
} mmodel_t;
```

Ao invés de:

```
class Pessoa {  
  nome: string;  
  idade: number;  
  altura: number;  
  peso: number;  
}
```


Pode-se usar:

```
class Pessoa {  
    public    String nome;  
    protected Integer idade;  
    protected Double altura;  
    private  Double peso;  
}
```

Indentação

A indentação tem como propósito demonstrar visualmente a estrutura hierárquica do código.

A maioria dos editores de texto sabe como indentar o código de acordo com a linguagem e as estruturas utilizadas.

```
<div class="">
  <div class="">
    <div class="">
      <div class="">
        <div class="">
          <div class="">
            <div class="">
              <div class="">
                <div class="">
                  <div class="">
                    <p></p>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Ao invés de:

```
class Pessoa {  
  nome: string;  
  idade: number;  
  altura: number;  
  
  constructor(nome: string, idade: number, altura: number) {  
    this.nome = nome;  
    this.idade = idade;  
  
    if (altura > 0) {  
      this.altura = altura;  
    } else {  
      this.altura = 0;  
    }  
  }  
}
```

Use:

```
class Pessoa {  
  nome: string;  
  idade: number;  
  altura: number;  
  
  constructor(nome: string, idade: number, altura: number) {  
    this.nome = nome;  
    this.idade = idade;  
  
    if (altura > 0) {  
      this.altura = altura;  
    } else {  
      this.altura = 0;  
    }  
  }  
}
```



```
4 <script>
5
6 let calc = () => {
7   const arr = [5, 4, 3, 2, 1].reduce((prev, cur) => prev + cur);
8   document.write (arr);
9 }
10
11 calc();
12
13 </script>
```

Escopos Minúsculos

Algumas estruturas possuem escopos muito pequenos, que podem ser facilmente reduzidos a uma única linha.

```
while(dis.read(buffer, 0, readBufferSize) != -1)
;
```

Ao invés de:

```
lista.map(item => {  
  item.id  
});
```

Use:

```
lista.map(item => item.id);
```

Essa indentação por muitas vezes não é bem entendida por programadores que não estão acostumados com a sintaxe.

Outro ponto importante é que facilmente pode-se ultrapassar o limite de caracteres por linha previamente definido.

Regra de Equipes

Equipes devem estabelecer padrões de formatação para que todos os membros da equipe possam entender o código.

Mesmo que tenha que escrever um código diferente do padrão que está acostumado, é importante focar no sucesso do time.



Material de Apoio

- [Baeldung - Formatting](#)
- [Hack.md](#)