

**CEUNSP**  
Centro Universitário  
N. Sra. do Patrocínio

Wellison da Cruz Bertelli - RGM: 5033482004

Pedro Henrique Lopes Siqueira - RGM: 5033490252

**ADS.1º Semestre**

**Prova A2 - “Gerador e Leitor de QR Codes com linguagem C”**

Gerando QR Codes utilizando a biblioteca em C Libqrencode v4.1.1 e escaneando com a biblioteca Web react-qrcode no Frontend (ReactJs).

Trabalho apresentado ao curso de Programação de Computadores sob supervisão do Profº Pagotto como requisito parcial para obtenção de nota semestral.



**CEUNSP – Salto 01/2023 CST Análise e Desenvolvimento de sistemas.  
Programação de Computadores I – Linguagem C.**

**Prova A2 – QR Code Generator em C + Apresentação WEB em Reactjs.**

**QR Code Generator em C:**

QR Codes são imagens 2D de matrizes binárias, formadas por pontos brancos e pretos que representam a informação, não apenas de URL's como qualquer texto ou objetos. A lógica inicial é sempre quebrar o problemão em pequenos problemas, e ir ticando cada um deles, ou seja, a primeira coisa é ir atrás de uma biblioteca especializada para trabalhar com gerações de QRCode na linguagem C.

Relatos pessoal:

Aí morava o problema, devemos sim valorizar a linguagem pois ela é a base de outras mas a verdade é que não possuem muitas documentações ou problemas resolvidos com ela em português Brasil, os problemas que ela se propõe é mais para otimizações com ganhos de performance, como desenvolver drivers, sistemas operacionais ou sistemas que iram rodar em hardwares simples como controladores e etc pois esse é o propósito do C. Não foi a melhor escolha de linguagem para resolução do nosso problema mas como é regra ter que utilizar ela, não desisti fácil e fui atrás! Após algum esforço encontrei uma biblioteca especializada em trabalhar com QRCode, **porém o C não ajuda muito por não possuir gerenciador de dependências na sua instalação padrão**, então novamente temos que perder tempo configurando ambiente, pois grande parte do processo foi isso, pouco desenvolvimento útil e muita configuração de ambiente.

**Configuração de ambiente:**

Existe um **gerenciador de dependências** muito conhecido no mundo C sendo ele o "Conan", **porém** como um grande entusiata GNU que eu sou descobri que **essa biblioteca está no repositório padrão do Ubuntu** e muitos outros sistemas unix like, pois grande parte dos programas nele são escritos em C, ou seja, temos grande parte das dependências da linguagem no repositório padrão pois grande parte da biblioteca Linux é escrita em C então podemos simplesmente instalar ela via apt pelo terminal.

```
$ sudo apt-get install libqrencode
```

Pronto, agora basta realizar os includes e utilizar as funcionalidades prontas para resolver nosso problema, NÃO CALMA ainda pode melhorar, se existe biblioteca existe algum programa pronto especializado nesse problema, e melhor ainda DISPONÍVEL NO REPOSITÓRIO, então basta instalarmos e gerarmos o QRCode tudo via terminal.

**Hello Word da biblioteca qrencode:**

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[1] + Done "/usr/bin/gdb" --interpreter=mi --tty=${DbgTe
soft-MIEngine-Out-4espfexh.22j"
welbert@pacotinho:~/Documentos/github/QRCodeReader$ SummarizingInvoiceNotes$
```

## Compilando no shell bash:

Devemos informar ao compilador gcc para incluir a biblioteca utilizada na hora de compilar o hello.c e gerar o executável com o seguinte cmd:

```
$ cd ./generatedQRCodeWithLibqrencode
```

```
$ gcc -o main hello.c -lqrencode
```

```
$ ./main
```

## Compilando no Visual Studio Code:

Devemos editar o task.json do ./vscode/task.json com o parâmetro: "-lqrencode" em args, sendo assim o arquivo task.json final deve ser algo como:

```
.vscode > {} tasks.json > [ ] tasks > -{} 0 > [ ] args > 11
1  {
2      "tasks": [
3          {
4              "type": "cppbuild",
5              "label": "C/C++: gcc-12 build active file",
6              "command": "/usr/bin/gcc-12",
7              "args": [
8                  "-fdiagnostics-color=always",
9                  "-g",
10                 "${file}",
11                 "-o",
12                 "${fileDirname}/${fileBasenameNoExtension}",
13                 "-lqrencode",
14                 "-lpng",
15                 "-lcjson",
16                 "-lrt",
17                 // "-fsanitize=address",
18                 // "-fsanitize=undefined",
19                 "-lcurl",
20                 "-lssl",
21                 "-lcrypto"
22             ],
23             "options": {
24                 "cwd": "${fileDirname}"
25             },
26             "problemMatcher": [
27                 "$gcc"
28             ],
29             "group": {
30                 "kind": "build",
31                 "isDefault": true
32             },
33             "detail": "Task generated by Debugger."
34         }
35     ],
36     "version": "2.0.0"
37 }
38
```

## Solução 02 Intermediário – Gerando output no console com UTF-8 e 3 quadrados dos cantos:

### Adicionando 3 quadrados dos cantos “*position detection patterns*”:

Position Detection Patterns (PDPs) são padrões de três quadrados localizados nos cantos do QR Code. Eles são usados para localizar e orientar o código para que possa ser corretamente lido pelo scanner. Cada PDP é composto por um quadrado maior cercado por três quadrados menores, formando um padrão simétrico. O tamanho e a posição dos PDPs são fixos e definidos pelo padrão QR Code, garantindo que qualquer scanner possa facilmente localizá-los e usá-los para corrigir a perspectiva e ler o código corretamente. Além disso, os PDPs também ajudam a diferenciar o código QR de outras imagens que possam estar presentes no mesmo local, garantindo que o scanner capture apenas o código QR desejado.

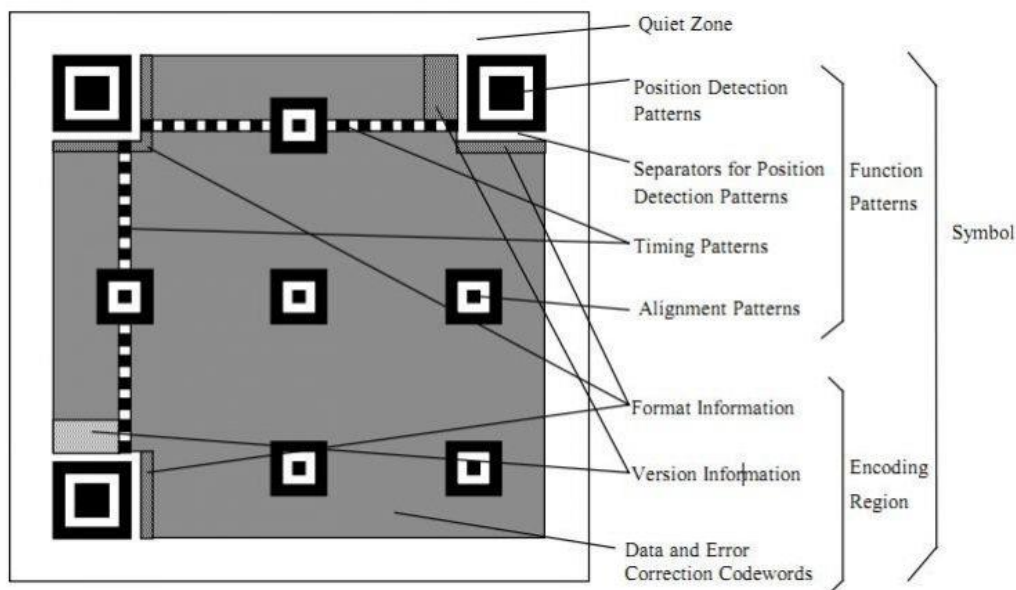
### Como funciona a dimensão de um QR Code:

Em primeiro lugar, digamos que existam 40 tamanhos de códigos QR. A versão oficial chama-se Versão. A versão 1 é uma matriz de 21 x 21, a versão 2 é uma matriz de 25 x 25 e a versão 3 tem tamanho de 29. Cada vez que uma versão é adicionada, o tamanho de 4 será aumentado. A fórmula é:  $(V - 1) * 4 + 21$  ( V é o número da versão) A versão mais alta 40,  $(40 - 1) * 4 + 21 = 177$ , então a mais alta é um quadrado de 177 x 177.

O Tamanho é dinâmico e aumenta e se dimensiona de acordo com o tamanho ou a quantidade de informações que seram armazenadas neste código, quanto maior essa quantidade de informações, menor será o volume de pixels e maior a quantidade deles. Porisso devemos nos atentar ao tamanho da informação que desejamos armazenar neste código pois a medida que ele aumenta mais difícil vai ser para o dispositivo escanear caso a camera não seja muito boa.

### Vejamos um exemplo de código QR:





### Composição resumida:

- **Bordas:** Neste caso é a marcação branca mais externa, e internamente está o código.
- **3 Quadrados indicadores:** Quadrado maior com outro menor internamente, utilizado para o dispositivo que está escaneando identificar que se trata de um QR Code e ignorar outras informações captadas juntamente com a imagem do QR propriamente dito.
- **Payload:** Carga útil, pontos pretos e brancos que representam a informação encriptada.

### Detalhando essa composição que forma o QR Code e implementação:

#### Padrão de posicionamento: (3 Quadrados dos cantos):

A função dele é ajudar o dispositivo que for escanear, a identificar e diferenciar o código QR do restante da imagem da câmera. Além disto eles também são uteis para ajudar no correto posicionamento deste processo.

**Position detection patterns, Padrão de detecção de posição** é um padrão de posicionamento usado para marcar o tamanho retangular do código QR. Esses três padrões de posicionamento têm bordas brancas chamadas Separadores para Padrões de Detecção de Posição (**Separators for Postion Detection Patterns**). A razão pela qual três em vez de quatro significa que três podem identificar um retângulo.

**Timing Patterns, Padrões de temporização** também são usados para posicionamento. O motivo é que existem 40 tamanhos de códigos QR e, se o tamanho for muito grande, é necessária uma linha padrão; caso contrário, pode ser digitalizado torto durante a digitalização.

**Alignment Patterns, Padrões de alinhamento** Somente os códigos QR da versão 2 ou superior (incluindo a versão 2) precisam desse material, que também é usado para posicionamento.

Esses padrões são utilizados para ajudar os dispositivos de leitura a identificar a posição e a orientação correta do QR Code.

- No código abaixo ele é representado pelo caractere `#` do printf ilustrado abaixo:

```
printf("%c", isPD ? '#' : (data[i * width + j] ? '1' : '0'));
```

### Dados Funcionais:

**Format Information**, **Informações de formato** existem em todos os tamanhos e são usadas para armazenar alguns dados formatados.

**Version Information**, Se as informações da versão forem **>= versão 7 ou superior**, duas áreas de 3 x 6 precisam ser reservadas para armazenar algumas informações da versão.

- No código abaixo não contém nada sobre pois a biblioteca lida com isso internamente por nós.

### Códigos de dados e códigos de correção de erros:

Além dos locais mencionados acima, os locais restantes armazenam o código de dados (payload) do **Data Code** e o código de correção de erro **Error Correction Code**.

- No código ele é o argumento: `QR\_ECLEVEL\_L` da função ilustrada abaixo:

```
QRcode *qrcode = QRcode_encodeString(string, 0, QR_ECLEVEL_L, QR_MODE_8, 1);
```

### Codificação de dados:

Existem vários algoritmos de codificações existentes, eu escolhi por utilizar o **Byte Mode** descrito logo abaixo pois ele é bem simples e a saída é UTF8, ou seja, podemos utilizar no stdout padrão (prompt/console).

Vamos falar primeiro sobre codificação de dados. O código QR suporta as seguintes codificações:

**Numeric Mode**, Modo numérico Codificação numérica, de 0 a 9. Se o número de dígitos a serem codificados não for um múltiplo de 3, então os últimos 1 ou 2 dígitos restantes serão convertidos em 4 ou 7 bits, e cada outro 3 dígitos serão codificados em 10, 12, 14 bits. O comprimento depende de o tamanho do código QR (há uma tabela abaixo da Tabela 3 para ilustrar este ponto).

**Alphanumeric mode**, Codificação de caracteres no modo alfanumérico. Inclui 0-9, letras maiúsculas de A a Z (sem letras minúsculas) e os símbolos \$ % \* + - . / : incluindo espaços. Esses caracteres são mapeados em uma tabela de índice de caracteres. Conforme mostrado abaixo: (onde SP é um espaço, Char é um caractere e Value é seu valor de índice) O processo de codificação consiste em agrupar os caracteres em pares, depois convertê-los no sistema de 45 bases da tabela abaixo e depois convertê-los em binário de 11 bits. Se houver um único, ele será convertido em binário de 6 bits. O modo de codificação e o número de caracteres precisam ser compilados em 9, 11 ou 13 números binários de acordo com diferentes tamanhos de versão (Tabela 3 na tabela a seguir):



| Char. | Value | Char. | Value | Char. | Value | Char. | Value | Char. | Value | Char. | Value | Char. | Value | Char. | Value |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 6     | 6     | C     | 12    | I     | 18    | O     | 24    | U     | 30    | SP    | 36    | .     | 42    |
| 1     | 1     | 7     | 7     | D     | 13    | J     | 19    | P     | 25    | V     | 31    | \$    | 37    | /     | 43    |
| 2     | 2     | 8     | 8     | E     | 14    | K     | 20    | Q     | 26    | W     | 32    | %     | 38    | :     | 44    |
| 3     | 3     | 9     | 9     | F     | 15    | L     | 21    | R     | 27    | X     | 33    | *     | 39    |       |       |
| 4     | 4     | A     | 10    | G     | 16    | M     | 22    | S     | 28    | Y     | 34    | +     | 40    |       |       |
| 5     | 5     | B     | 11    | H     | 17    | N     | 23    | T     | 29    | Z     | 35    | -     | 41    |       |       |

**Byte mode**, Modo de byte, codificação de byte, pode ser caracteres ISO-8859-1 de 0-255. Alguns scanners de código QR podem detectar automaticamente se é codificação UTF-8.

**Todos os modos disponíveis:** Numeric mode, Alphanumeric mode, Byte mode, Kanji mode, Extended Channel Interpretation (ECI) mode, Structured Append mode, FNC1 mode) não irei abordar todos esses modos, pois ficaria de mais.

Esses algoritmos são utilizados para codificação e assim temos diferentes tipos de representações possíveis, podemos representar o QR Code com 0's e 1's, podemos representar como imagem, podemos representar como números decimais, podemos representar como caracteres especiais, podemos representar de várias formas, no caso do código abaixo representei o payload com 0's e 1's pois utilizamos o tipo **Byte mode** no argumento (Poderíamos também representar com o valor RGB nas cores preto (000) ou branco (FFF) ao invés de 0's e 1's e assim gerar stdout colorido).

- No código ele é o argumento: `QR_MODE_8` da imagem ilustrativa abaixo:

```
QRcode *qrcode = QRcode_encodeString(string, 0, QR_ECLEVEL_L, QR_MODE_8, 1);
```

### Algoritmo completo:

Criamos o objeto principal QRCode (que basicamente é uma matriz bidimensional contendo o payload e todo o resultado dos processamentos de correções de erros e etc) criado apartir de uma String `"hello, world!"`, dizemos que queremos trabalhar com UTF8 com o argumento `QR_MODE_8` logo o valor contido dos payloads na matriz será representado por `0's e 1's` e os 3 quadrados dos cantos serão representados por `"#"`.

Após criado o objeto, podemos percorrer a matriz bidimensional em um looping for e realizar as devidas análises para saber se o elemento corrente é payload(Carga útil)/correção de erros(bit redundantes) representados no stdout como `0's e 1's` ou se é os 3 quadrados dos cantos (Position detection patterns) que será utilizado como representação o caractere especial `"#"`.



- No código essas verificações são feitas no looping for da imagem ilustrativa abaixo:

```
QRcode int width = qrcode->width;
unsigned char *data = qrcode->data;

// Imprime a matriz do QR code com os quadrados de Position Detection Patterns
int i, j;
for (i = 0; i < width; i++) {
    for (j = 0; j < width; j++) {
        // Verifica se a posição corresponde a um quadrado de Position Detection Pattern
        int isPD = (i < 7 && j < 7) || // Canto superior esquerdo
                   (i < 7 && j >= width - 7) || // Canto superior direito
                   (i >= width - 7 && j < 7); // Canto inferior esquerdo

        // Imprime '#' se for um quadrado de Position Detection Pattern, e caso contrário,
        // faz outra verificação para saber se o elemento corrente no looping for é o
        // correspondente do payload (carga útil que é a informação em si + bits
        // redundantes para correções de erros ditos anteriormente) e assim imprime o
        // correspondente 0 ou 1 no console:
        printf("%c", isPD ? '#' : (data[i * width + j] ? '1' : '0'));
    }
    printf("\n");
}
```

### Imagem ilustrativa dos códigos e conceitos ditos anteriormente:

Função que imprime a matriz bidimensional no console (view):

```
6 void printQRcode(QRcode *qrcode) {
7     int width = qrcode->width;
8     unsigned char *data = qrcode->data;
9
10    // Imprime a matriz do QR code com os quadrados de Position Detection Patterns
11    int i, j;
12    for (i = 0; i < width; i++) {
13        for (j = 0; j < width; j++) {
14            // Verifica se a posição corresponde a um quadrado de Position Detection Pattern
15            int isPD = (i < 7 && j < 7) || // Canto superior esquerdo
16                      (i < 7 && j >= width - 7) || // Canto superior direito
17                      (i >= width - 7 && j < 7); // Canto inferior esquerdo
18
19            // Imprime '#' se for um quadrado de Position Detection Pattern, e caso contrário,
20            // faz outra verificação para saber se o elemento corrente no looping for é o
21            // correspondente do payload (carga útil que é a informação em si + bits redundantes
22            // para correções de erros ditos anteriormente) e assim imprime o correspondente
23            // 0 ou 1 no console:
24            printf("%c", isPD ? '#' : (data[i * width + j] ? '1' : '0'));
25        }
26        printf("\n");
27    }
28 }
```

Função principal (main) que cria o objeto principal QRCode da biblioteca e fornece como argumento para a função responsável pela apresentação (stdout prompt):

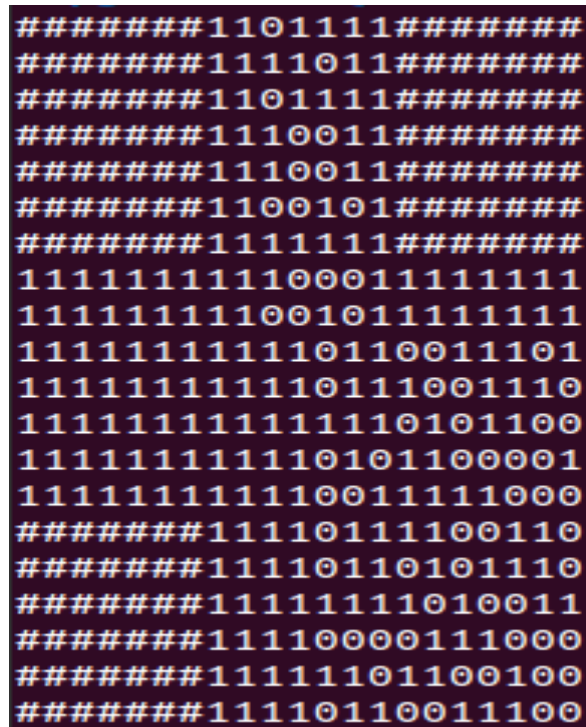
```
30 int main() {
31     char string[] = "Hello, world!";
32     QRcode *qrcode = QRcode_encodeString(string, 0, QR_ECLEVEL_L, QR_MODE_8, 1);
33
34     // Imprime o QR code com os Position Detection Patterns
35     printQRcode(qrcode);
36
37     QRcode_free(qrcode);
38
39     return 0;
40 }
41
```

Output (stdout) no console prompt Bash:

```
$ cd ./QRCodeReader_SummarizingInvoiceNotes/generatedQRCodeWithLibqrencode
```

```
$ gcc -o helloFinal ./helloAddQuadradosDosCantos_ByteMode-UTF8.c -lqrencode
```

```
$ ./helloFinal
```



**Bônus:** Frontend em ReactJs para escanear QR Codes gerados e obter o resultado da descrição de QR Codes utilizando a biblioteca react-qr-reader, como não é o escopo principal do trabalho que é sobre a linguagem C não vai conter explicações sobre a implementação dessa camada de apresentação aqui neste documento, mas para qualquer dúvida segue repositório com códigos abertos e documentações sobre o processo de desenvolvimento:

[https://github.com/WelBert-dev/QRCodeReader\\_SummarizingInvoiceNotes](https://github.com/WelBert-dev/QRCodeReader_SummarizingInvoiceNotes)

## Referências:

- Documentação base em Japonês:

<https://coolshell.cn/articles/10590.html>

- Repositório Oficial do Libqrencode no github:

<https://github.com/fukuchi/libqrencode>

- Espelho Repositório Oficial da biblioteca libqrencode no Ubuntu:

<https://packages.ubuntu.com/search?keywords=libqrencode-dev>

- Espelho Repositório Oficial do utilitário em linha de comando Bash:

<https://manpages.ubuntu.com/manpages/jammy/man1/qrencode.1.html>