



Module C/C++

Génération d'un code-barres et d'un QRcode en bitmap



Table des matières

Objectifs et ressources.....	2
- Défi 1 – Les éléments EAN.....	3
- Défi 2 – Le code binaire du code-barres.....	5
- Défi 3 – Génération de l'image du code-barres.....	8
- Défi 4 – Test d'une bibliothèque permettant la génération d'un QRcode.....	10
- Défi 5 – Génération de l'image du QRcode.....	14
Annexe 1 : la classe SNIImage.....	16
Annexe 2 : bibliothèque QRcode qrcodegen.h.....	17

Objectifs et ressources

Objectifs

Étudier le codage d'un code-barres afin de générer, dans une console, la séquence binaire correspondant à un code donné. Calcul du chiffre de contrôle. Génération d'une image du code-barres avec incorporation des chiffres du code. Utiliser une bibliothèque écrite en langage C afin d'afficher dans une console le dessin d'un QRcode correspondant au texte saisi par l'utilisateur. La version du QRcode s'adapte automatiquement à la taille du message. Génération d'une image du QRcode. Dans un premier temps, le code sera programmé en langage C, mis à part l'utilisation de **cout** de la classe C++ **iostream** pour les affichages. Pour un code 100 % langage C, utiliser **printf()** à la place de **cout**. Dans un second temps, la classe SNIImage en C++ sera utilisée pour générer une image bitmap 24 bits.

- Tableaux de caractères
- Tableaux à 2 dimensions
- Langage C et C++
- Algorithmie
- Binaire
- Utilisation d'une classe de traitement d'image

Logiciels

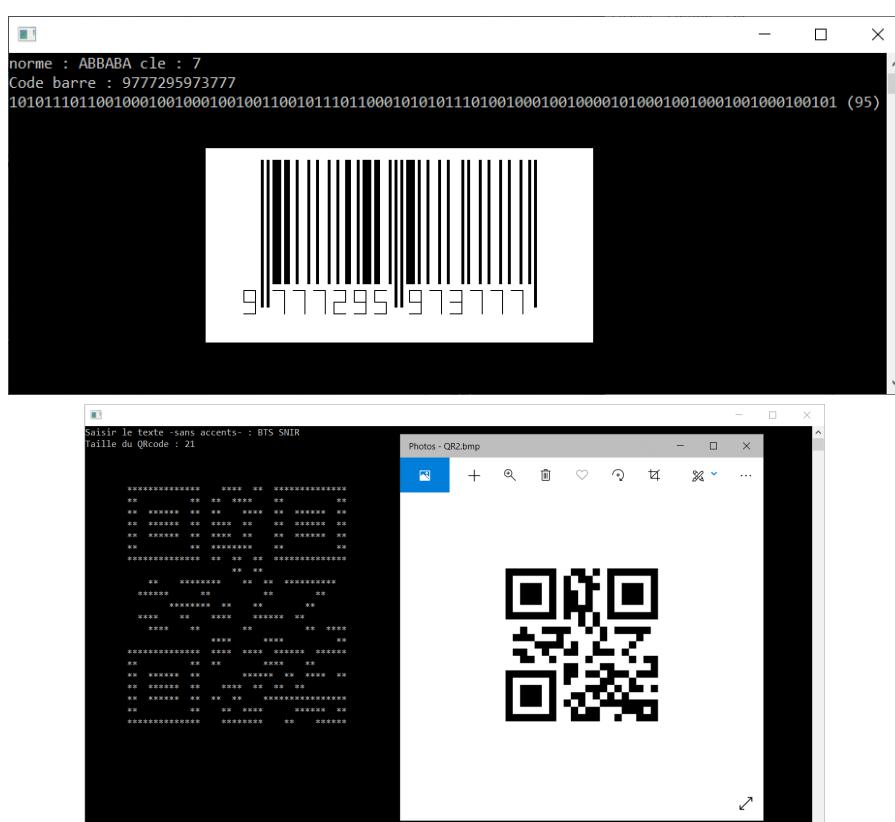
- C++ Builder
- CodeBlocks

Annexes

- La classe SNIImage
- Bibliothèque QRCode qrcodegen.h

Ressources

- SNIImage.h
- SNIImage.o (pour CodeBlocks)
- SNIImage.obj (pour C++ Builder)
- qrcodegen.h, qrcodegen.c



– Défi 1 – Les éléments EAN

Le code EAN (European Article Numbering ; aussi appelé International Article Number ou IAN) est un code-barres utilisé par le commerce et l'industrie conformément aux spécifications d'EAN International.

Les éléments EAN se caractérisent par une succession de quatre barres (deux barres claires qui alternent avec deux barres sombres) dont la somme des largeurs vaut toujours 7 modules. Il y a donc au total 7 barres élémentaires dans un élément. Chacune de ces barres est elle-même constituée de la juxtaposition de 1 à 4 barres élémentaires de même couleur. Source : https://fr.wikipedia.org/wiki/Code-barres_EAN.

Les trois types d'éléments

Chaque élément peut être représenté en binaire par une suite de 7 bits : un X ou 1 correspondant à une barre élémentaire noire, un _ ou 0 correspondant à une barre élémentaire blanche. Voici les représentations des 10 chiffres comme éléments A, B ou C :

élément A	élément B	élément C	élément A	élément B	élément C
0 [____XX_X]	[_X__XXX]	[XXX__X_]	[_XX]	[_XX]	[_XX]
1 [__XX__X]	[_XX__XX]	[XX__XX]	[XX]	[XX]	[XX]
2 [_X_X__XX]	[_XX_X__XX]	[XX_X__X]	[XX]	[XX]	[XX]
3 [_XXXX_X]	[_X__X_X]	[X__XX_]	[XX]	[XX]	[XX]
4 [_X__XX]	[_XXX_X_X]	[X_X__XX]	[XX]	[XX]	[XX]
5 [_XX__X_X]	[_XXX_X_X]	[X_X__XXX]	[XX]	[XX]	[XX]
6 [_X_XXXX]	[___X_X]	[X_X__]	[XX]	[XX]	[XX]
7 [_XXX_XX]	[_X_X__]	[X_X__]	[XX]	[XX]	[XX]
8 [_XX__XXX]	[_X_X__]	[X_X__]	[XX]	[XX]	[XX]
9 [___X_XX]	[_X__XXX]	[XXX_X__]	[XX]	[XX]	[XX]

soit,
graphiquement

A partir du codage des 3 éléments détaillés ci-dessus, il est nécessaire de créer 3 tableaux à 2 dimensions (**binaireA**, **binaireB**, **binaireC**) permettant de stocker 10 chaînes de caractères comportant chacune 7 caractères représentant les 7 bits de l'élément EAN :

//début de code :

```
char binaireA[10][8]={"0001101","0011001", /* à compléter */};  
/* ... */
```

Ainsi **binaireA[0]** contiendra "0001101", c'est à dire le code du **0** pour l'élément de type A...

Une variable entière **chiffre** permettra de stocker le chiffre saisi par l'utilisateur.



Donner le code permettant de saisir le chiffre et d'afficher les 3 éléments EAN correspondant.

Coder et tester le programme en mode console.

//exemple de sortie console :

```
Entrer un chiffre : 1  
Element de type A : 0011001  
Element de type B : 0110011  
Element de type C : 1100110
```

La zone de garde normale et centrale

Le code-barres est encadré par une zone de garde dite "normale". Une zone de garde dite "centrale" permet de séparer le code-barres en 2 ensemble de 6 éléments EAN (pour le code EAN 13).

Il faut créer 2 tableaux de caractères : **zoneNormale** et **zoneCentrale**, initialisés selon la norme suivante :

zone de garde normale : **"101"**

zone de garde centrale : **"01010"**

 Coder et tester le programme permettant d'afficher les codes des zones de garde.

//exemple de sortie console :

```
Zone de garde normale : 101
```

```
Zone de garde centrale : 01010
```

Stockage et affichage des 13 chiffres du code EAN-13

Un tableau de caractères **codeBarres** sera chargé de stocker les 13 chiffres du code. Il faudra initialiser ce tableau à la valeur d'un code-barres existant ("3401312345624" par exemple). Avec la caractère de fin de chaîne '\0', le nombre de caractères maximum de ce tableau sera **14**.

 Coder et tester le programme permettant d'afficher les 13 chiffres.

//exemple de sortie console :

```
Le code : 3401312345624
```

– Défi 2 – Le code binaire du code-barres

Le premier chiffre du code-barres

le code barre débute et se termine par une zone de garde normale. Le premier chiffre entraîne une séquence précise d'éléments EAN : "le motif". Suivra une zone de garde centrale puis 6 éléments de type C. Voici le tableau donnant la correspondance entre le motif et le 1er chiffre :

1 ^{er} chiffre	Motif	Remarques
0	[AAAAAA]	
1	[AABABB]	
2	[AABBAB]	
3	[AABBBA]	Parmi les 64 combinaisons possibles de 6 éléments de type A ou B (avant la zone de garde centrale), <ul style="list-style-type: none"> • seules 10 combinaisons sont utilisées pour coder un chiffre ; • les motifs ont tous un élément de type A, toujours présent en première position ; • si un élément de type B est présent, il y a exactement trois éléments de ce type dans le motif ; • les autres combinaisons ne sont pas utilisées.
4	[ABAABB]	
5	[ABBAAB]	
6	[ABBBAA]	
7	[ABABAB]	
8	[ABABBA]	
9	[ABBABA]	

Un tableau de caractères à 2 dimensions **motif** permettra de stocker les 10 motifs de 6 caractères :

//début de code :

```
char motif[10][7]={"AAAAAA", "AABABB", /* à compléter */};
```

Une variable entière **premierChiffre** permettra de stocker le premier élément du tableau **codeBarres[0]** auquel il faudra enlever la valeur ASCII du caractère '0'. En effet le premier caractère '0' (code ASCII **0x30**) doit correspondre au chiffre **0**. Il sera alors possible d'afficher l'élément d'indice **premierChiffre** du tableau **motif** : **motif[premierChiffre]**.



Donner le programme permettant d'afficher le motif correspondant au premier chiffre du code.



Coder et tester le programme.

//exemples de sortie console :

```
Le code : 3401312345624
Motif : AABBBA
```

```
Le code : 5837478696658
Motif : ABBAAB
```

Le code binaire complet

Il faut maintenant afficher le code binaire complet du code-barres. Un tableau de caractères `codeBinaireComplet` permettra de stocker le nombre binaire au format ASCII (0 sera représenté par '0', et 1 par '1'). Il sera initialisé à "" (chaîne vide).



Au vue de la norme, quelle est la taille minimum de ce tableau ?

En utilisant la fonction `strcat` (<http://www.cplusplus.com/reference/cstring/>), il faut placer dans le tableau `codeBinaireComplet` le code de début (`zoneNormale`). Pour simplifier la visualisation lors des tests, un séparateur "-" sera ajouté. Une variable entière `chiffre` permettra de stocker le deuxième élément du tableau `codeBarres[1]` auquel il faudra enlever la valeur ASCII du caractère '0'. Pour ajouter le deuxième chiffre, il faut procéder ainsi : si `motif[Chiffre]` est un 'A', il faut ajouter à `codeBinaireComplet` le code `binaireA[chiffre]`. Si c'est un 'B', il faut ajouter `binaireB[chiffre]`.



Donner le programme permettant d'ajouter le 2ème chiffre au code binaire, puis d'afficher le code.



Coder et tester le programme.

//exemples de sortie console :

```
Le code : 5837478696658
Motif : ABBAAB
101-0110111
```



Donner le programme permettant :

- d'ajouter de la même manière les 5 chiffres suivants correspondant à un motif 'A' ou 'B'.
- d'ajouter le code de la zone centrale.
- d'ajouter enfin les 6 derniers chiffres du code-barres : ce sont des éléments de type 'C'.
- de terminer par le code de fin (`zoneNormale`).



Coder et tester le programme permettant d'afficher le code binaire complet.

//exemple de sortie console :

```
Le code : 5837478696658
Motif : ABBAAB
101-0110111-0100001-0010001-0100011-0111011-0001001-01010-
1010000-1110100-1010000-1010000-1001110-1001000-101
```

Vérification du chiffre de contrôle

Le dernier chiffre d'un code-barres est un chiffre permettant de vérifier la validité du code-barres. Pour obtenir ce chiffre, il faut calculer trois fois la somme des chiffres de rang pair (en partant du second chiffre) de gauche à droite, cette valeur sera stockée dans un entier **sommeImpairs**. Dans notre exemple (5837478696658) $\text{sommeImpairs} = (8+7+7+6+6+5)*3$. Il faut calculer ensuite la somme des chiffres de rang impair (en partant du premier chiffre) de gauche à droite, et stocker cette valeur dans **sommePairs**. Dans notre exemple : $\text{sommePairs} = 5+3+4+8+9+6$. Le total de ces deux sommes partielles vaut alors **152**. L'entier **R** est le chiffre des unités de la somme partielle, **R** est obtenu en utilisant un modulo 10 (reste de la division par 10) : $\text{R} = 152 \% 10 = 2$. La valeur de la clé est alors **(10-R) modulo 10** : $(10-2) \% 10 = 8$.



Donner le code permettant de mettre en œuvre l'algorithme précédent.



Coder et tester le programme permettant de calculer la clé de contrôle.

//exemple de sortie console :

```
Le code : 5837478696658
Motif : ABBAAB
101-0110111-010001-0010001-0100011-0111011-0001001-01010-
1010000-1110100-1010000-1010000-1001110-1001000-101
cle : 8
```

/OPTION

Génération d'un code aléatoire

En utilisant les fonctions **rand()** et **srand()**, générer les **12** premiers chiffres du code-barres de manière aléatoire. Calculer la clé puis ajouter cette clé à la suite du code. **srand(time(0))** permet d'initialiser la fonction **rand**, **rand()%10** génère un chiffre aléatoire de **0 à 9** (<http://www.cplusplus.com/reference/cstdlib/rand/>).



Coder et tester le programme permettant de générer un code-barres aléatoire.

– Défi 3 – Génération de l'image du code-barres

La classe **SNImage** permet de dessiner dans une image bitmap 24 bits (.bmp). Le diagramme de classe est donné en annexe. Sont donnés également : le fichier .h , le .o (ou le .obj) : ce dernier est le code des méthodes de la classe déjà compilé, il dépend d'un compilateur.

 Placer le fichier codebarres.bmp (une image blanche de 400x200 en bitmap 24 bits créée avec Paint par exemple) dans :

- Win32\Debug pour C++ Builder,
- bin\Debug pour CodeBlocks, ainsi que dans le même répertoire que votre code source.

Placer :

- SNImage.obj dans Win32\Debug pour C++ Builder
- SNImage.o dans obj\Debug pour CodeBlocks.

Dans l'environnement de développement choisi, ajouter le fichier (o ou .obj) au projet :

- C++ Builder : clic droit sur Project1.exe dans le gestionnaire de projet > Ajouter... Win32\Debug\SNImage.obj
- CodeBlocks : Settings > Compiler and debugger... > Linker settings > Add obj\Debug\SNImage.o

Test de la classe SNImage

Dans le fichier du programme principal, inclure **SNImage.h**. Dans le main, créer un objet **img** de la classe **SNImage**. Appeler la méthode **Charger**, en passant "codeBarres.bmp" en argument, afficher la largeur et la hauteur de l'image en appelant les méthodes Largeur() et Hauteur().

 Tester le programme et vérifier les dimensions affichées.

Ajouter la sauvegarde dans une autre image en appelant la méthode **Sauvegarde** et en passant "codeBarresResultat.bmp" en argument. Après exécution le nouveau fichier est créé dans le même répertoire que l'image d'origine. Avant la sauvegarde, appeler -seulement pour faire un test- la méthode **Negatif()**. L'image créée après exécution devient noire.

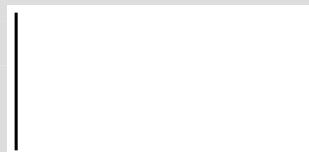
 Tester le programme et vérifier la présence de la nouvelle image.

Tracé des barres

La classe **SNImage** utilise 2 structures : **Pixel** et **Coordonnee**. **Pixel** est composé des 3 composantes d'un pixel : **rouge**, **vert** et **bleu** ce sont des octets. **Coordonnee** contient 2 entiers : **ligne** et **colonne**.

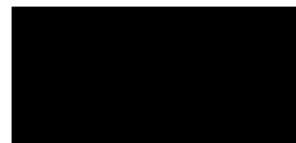
 Tester le code suivant : la première barre du code (180 x 4 pixels) est dessinée dans l'image résultat.

```
img.Charger("codeBarres.bmp");
Coordonnee debut;
Pixel noir;
noir.rouge=0;
noir.bleu=0;
noir.vert=0;
debut.ligne=10;
debut.colonne=10;
img.Colorier(debut,4,180,noir);
//void Colorier(Coordonnee coord,int l,int h,Pixel couleur);
img.Sauvegarder("codeBarresResultat.bmp");
```



Le code-barres est constitué de 95 barres.

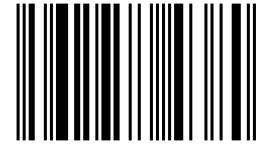
 Afin d'obtenir les 95 barres, appeler la méthode Colorier dans une boucle permettant 95 itérations et chargée d'ajouter 4 à debut.colonne à chaque itération.



Il ne reste plus qu'à ajouter deux conditions : la méthode **Colorier** n'est appelée que si **codeBinaireComplet[i]** est égal à '1'. La colonne **debut.colonne** est augmenté de 4 si **codeBinaireComplet[i]** est différent de '-' (le séparateur dans notre tableau).

 Donner les instructions nécessaires à ces deux conditions.

 Coder et tester le programme. Vérifier la génération du code-barres : il reste à le scanner avec un smartphone pour vérifier le code.



Ajout des chiffres sous le code-barres

Il est nécessaire de modifier la hauteur des barres à 150 pixels, puis de créer une nouvelle variable de type **Coordonnee** : **debutTexte** initialisée à 165 pour la ligne et 10 pour la colonne.

Appeler la méthode **void Dessiner7Segments(Coordonnee coord,int taille,int epaisseur,Pixel couleur,string message)**, permettant de dessiner des chiffres au format 7 segments sur une image : choisir une taille de 27, une épaisseur de 1, et dessiner le message **codeBarres**.

 Coder et tester le programme final.



//OPTION

Dessin des séparateurs

Modifier le programme afin d'obtenir des barres plus longues pour les zones de garde. Les chiffres doivent être décalés afin d'être lisibles de part et d'autre des zones de garde.



 Coder et tester le programme.

– Défi 4 – Test d'une bibliothèque permettant la génération d'un QRcode

QR (abréviation de l'anglais Quick Response) signifie que le contenu du code peut être décodé rapidement après avoir été lu par un lecteur de code-barres, un téléphone mobile, un smartphone, ou encore une webcam. Son avantage est de pouvoir stocker plus d'informations qu'un code à barres. Il existe différentes versions du code QR allant de 25 à 4296 caractères :

Version 1, 21×21 , 10-25 caractères.

Version 2, 25×25 , 20-47 caractères.

Version 3, 29×29 , 35-77 caractères.

Version 4, 33×33 , 67-114 caractères.

Version 10, 57×57 , 174 à 395 caractères.

Version 40, 177×177 , 1 852 à 4 296 caractères.



V1



V3



V10

Les codes QR utilisent le système Reed-Solomon pour la correction d'erreur : le code contient jusqu'à 30 % de redondance lui permettant d'être lisible même si une partie de l'image est perturbée. L'algorithme Reed-Solomon étant trop complexe (il relève plutôt d'un travail de chercheur universitaire), nous allons utiliser une bibliothèque open-source écrite en langage C (le fichier d'entête est donné dans l'annexe 2, la bibliothèque est fournie dans l'archive des ressources).

 Créer un nouveau projet en mode console. Ajouter au projet qrcodegen.c et inclure qrcodegen.h dans le fichier principal. Compiler le programme.

Saisie du texte à stocker dans le QRcode

On demande à l'utilisateur de saisir le texte, ce dernier sera stocké dans un tableau **texte** de 1000 caractères : il faudra utiliser la méthode **getline** de l'objet **cin** (le premier argument de la méthode est le texte, le deuxième est la longueur maximum à lire).

 Saisir et afficher le texte afin de vérifier qu'il a été correctement stocké dans le tableau. Ne pas oublier d'appeler la méthode **get** de l'objet **cin** afin que la fenêtre du programme ne se ferme pas.

//exemple de sortie console :

```
Saisir le texte -sans accents- : test de generation de QRcode
test de generation de QRcode
```

Utilisation de la bibliothèque de génération d'un QRcode

Nous allons utiliser la fonction :

```
bool qrcodegen_encodeText(const char *text, uint8_t dataAndTemp[], uint8_t qrcode[],  
enum qrcodegen_Ecc ecl, int minVersion, int maxVersion, enum qrcodegen_Mask mask,  
bool boostEcl);
```

Voici le descriptif présent dans le fichier source :

```
/* Encodes the given binary data to a QR Code, returning true if encoding succeeded.  
 * If the data is too long to fit in any version in the given range  
 * at the given ECC level, then false is returned.  
 * - The input array range dataAndTemp[0 : dataLen] should normally be  
 *   valid UTF-8 text, but is not required by the QR Code standard.  
 * - The variables ecl and mask must correspond to enum constant values.  
 * - Requires 1 <= minVersion <= maxVersion <= 40.  
 * - The arrays dataAndTemp and qrcode must each have a length  
 *   of at least qrcodegen_BUFFER_LEN_FOR_VERSION(maxVersion).  
 * - After the function returns, the contents of dataAndTemp may have changed,  
 *   and does not represent useful data anymore.  
 * - If successful, the resulting QR Code will use byte mode to encode the data.  
 * - In the most optimistic case, a QR Code at version 40 with low ECC can hold any byte  
 *   sequence up to length 2953. This is the hard upper limit of the QR Code standard.  
 * - Please consult the QR Code specification for information on  
 *   data capacities per version, ECC level, and text encoding mode. */
```

Voici les arguments de type **enum** (énumération) possibles :

```
/* The error correction level in a QR Code symbol. */  
enum qrcodegen_Ecc {  
    // Must be declared in ascending order of error protection  
    // so that an internal qrcodegen function works properly  
    qrcodegen_Ecc_LOW = 0, // The QR Code can tolerate about 7% erroneous codewords  
    qrcodegen_Ecc_MEDIUM, // The QR Code can tolerate about 15% erroneous codewords  
    qrcodegen_Ecc_QUARTILE, // The QR Code can tolerate about 25% erroneous codewords  
    qrcodegen_Ecc_HIGH, // The QR Code can tolerate about 30% erroneous codewords  
};
```

```
/* The mask pattern used in a QR Code symbol. */  
enum qrcodegen_Mask {  
    // A special value to tell the QR Code encoder to  
    // automatically select an appropriate mask pattern  
    qrcodegen_Mask_AUTO = -1,  
    // The eight actual mask patterns  
    qrcodegen_Mask_0,  
    qrcodegen_Mask_1,  
    qrcodegen_Mask_2,  
    qrcodegen_Mask_3,  
    qrcodegen_Mask_4,  
    qrcodegen_Mask_5,  
    qrcodegen_Mask_6,  
    qrcodegen_Mask_7,  
};
```

En étudiant cette documentation, donner un exemple d'appel de la fonction `qrcodegen_encodeText`.



Affichage de la taille du QRcode

Dans le programme principal, il faut créer 2 tableaux de caractères de taille `qrcodegen_BUFFER_LEN_MAX` : `buffer` et `qrCode`. La fonction `qrcodegen_encodeText` sera appelée avec les arguments :

- `texte`
- `buffer`
- `qrCode`
- `qrcodegen_Ecc_LOW`
- `1`
- `40`
- `qrcodegen_Mask_AUTO`
- `true`

La valeur de retour doit être stockée dans un booléen. Si cette valeur est vraie, appeler la fonction `qrcodegen_getSize` en lui passant le QRcode en argument (`qrCode`). Stocker la valeur de retour dans un entier : `tailleQR`, puis afficher cet entier.



Coder et tester le programme permettant de saisir un texte et d'afficher la taille du QRcode correspondant.

//exemples de sorties console :

```
Saisir le texte -sans accents- : test  
Taille du QRcode : 21
```

```
Saisir le texte -sans accents- : test un peu plus long  
Taille du QRcode : 25
```

```
Saisir le texte -sans accents- : test encore bien plus long que le precedent  
Taille du QRcode : 29
```

```
Saisir le texte -sans accents- : Il partit comme un trait ; mais les elans qu'il fit  
furent vains : la Tortue arriva la premiere. Eh bien, lui crie-t-elle, avais-je pas  
raison? De quoi vous sert votre vitesse ? Moi l'emporter ! et que serait-ce si vous  
portiez une maison ?  
Taille du QRcode : 57
```

Affichage du QRcode

La fonction `qrcodegen_getModule` permet de lire un pixel du QRcode :

```
bool qrcodegen_getModule(const uint8_t qrcode[], int x, int y);
```

Voici le descriptif présent dans le fichier source :

```
/* Returns the color of the module (pixel) at the given coordinates, which is false  
 * for white or true for black. The top left corner has the coordinates (x=0, y=0).  
 * If the given coordinates are out of bounds, then false (white) is returned. */
```

Il faut prévoir un rebord blanc de 4 pixels autour du QRcode : et créer une variable **rebord** initialisée à **4**. Dans une double boucle **for** (**y** allant de **-rebord** à **tailleQR + rebord** par valeur strictement inférieur, **y** allant de **un** en **un**, même chose pour **x**), appeler la fonction **qrcodegen_getModule**, en passant **qrCode**, **x** et **y** en argument. Stocker la valeur de retour dans un booléen **pix**. Si **pix** est à **1**, afficher "******", sinon afficher " ", ajouter un saut de ligne après la fin de la boucle interne en **x**. Le QRcode s'affiche dans la console, mais il n'est pas encore "scannable" : il faudra pour cela générer une image.



Coder et tester le programme permettant d'afficher la QRcode dans la console.

//exemple de sortie console :

```
Saisir le texte -sans accents- : test  
Taille du QRcode : 21
```

– Défi 5 – Génération de l'image du QRcode

Test de la classe **SNImage**

Reprendre le test de la classe **SNImage** présenté au début du défi 3, l'image de départ sera nommée "QR.bmp" (utiliser une image de 1000x1000 blanche en bitmap 24 bits), le résultat sera nommé "QRresultat.bmp".

Adaptation de la taille de l'image à la taille du QRcode

Pour obtenir une meilleure définition, un pixel sera représenté par un carré de 10x10 pixels. Il faudra créer un entier **taillePixel** initialisé à 10. Créer également un entier **tailleImage** initialisé à **taillePixel** multiplié par **tailleQR** augmenté par 2 fois le **rebord**. Créer une variable **init** de type **Coordonnee** dont la **ligne** et la **colonne** sont initialisés à 0. Après avoir chargé l'image **QR.bmp**, appeler la méthode **Recadrer** en passant en argument **init** puis **tailleImage** pour les arguments hauteur et largeur. Afficher la largeur et la hauteur de l'image résultante.



Donner le code permettant de mettre en œuvre l'algorithme précédent.



Coder et tester le programme permettant de recadrer l'image en fonction de la longueur du texte saisi.

//exemples de sortie console :

```
Saisir le texte -sans accents- : test
Taille du QRcode : 21
Taille de l'image : 290x290
```

```
Saisir le texte -sans accents- : test un peu plus long
Taille du QRcode : 25
Taille de l'image : 330x330
```

```
Saisir le texte -sans accents- : test encore bien plus long que le precedent
Taille du QRcode : 29
Taille de l'image : 370x370
```

```
Saisir le texte -sans accents- : Il partit comme un trait ; mais les elans qu'il fit
furent vains : la Tortue arriva la premiere. Eh bien, lui cria-t-elle, avais-je pas
raison? De quoi vous sert votre vitesse ? Moi l'emporter ! et que serait-ce si vous
portiez une maison ?
Taille du QRcode : 57
Taille de l'image : 650x650
```

Dessin du QRcode

Pour dessiner les parties noires dans l'image, nous utiliserons la méthode **Colorier** de la classe **SNIImage**. Pour cela nous avons besoin d'une variable **coordonneePix** de type **Coordonnee** et d'une variable **noir** de type **Pixel** dont chaque composante (rouge vert et bleu) est initialisée à **0** pour obtenir la couleur noire. Au cœur de la double boucle **for**, stocker **taillePixel*(y+rebord)** dans la composante **ligne** de **coordonneePix**. Dans la composante **colonne**, stocker **taillePixel*(x+rebord)**. Si **pix** est égal à **1**, appeler la méthode **Colorier** de l'objet **img** en passant en argument **coordonneePix**, **taillePixel** pour la hauteur et la largeur, puis **noir**. Ne pas oublier de **Sauvegarder** l'image dans "**QRresultat.bmp**" à la fin du programme.



Donner le code permettant de mettre en œuvre l'algorithme précédent.



Coder et tester le programme permettant de générer l'image du QRcode : l'image générée (**QRresultat.bmp**) peut être scannée avec une application sur smartphone.

//exemples de sortie console :

```
test
Taille du QRcode : 21
Taille de l'image : 290x290
```



```
test un peu plus long
Taille du QRcode : 25
Taille de l'image : 330x330
```



```
test encore bien plus long que le precedent
Taille du QRcode : 29
Taille de l'image : 370x370
```



Il partit comme un trait ; mais les elans qu'il fit furent vains : la Tortue arriva la premiere. Eh bien, lui cria-t-elle, avais-je pas raison? De quoi vous sert votre vitesse ? Moi l'emporter ! et que serait-ce si vous portiez une maison ?

```
Taille du QRcode : 57
Taille de l'image : 650x650
```



Annexe 1 : la classe SNIImage

SNIImage
<p><i>attributs</i></p> <pre> -dimensionMax : unsigned long -tailleFichier : unsigned long -debutImage : unsigned long -tailleImage : unsigned long -tailleEntete : unsigned long -zero : unsigned long -format : unsigned long -resoLarg : unsigned long -resoHaut : unsigned long #image : Pixel*" " #signature : unsigned short #largeur : unsigned long #hauteur : unsigned long </pre>
<p><i>opérations</i></p> <pre> -TracerCercleUnPixel(centre : Coordonnee, rayon : int, couleur : Pixel) : void -TracerDiagonaleUnPixel(debut : Coordonnee, fin : Coordonnee, couleur : Pixel) : void -ChangerCouleurSegment(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel, segment : char) : void +SNIImage() +~SNIImage() +Signature() : unsigned short +Largeur() : unsigned long +Hauteur() : unsigned long +TailleFichier() : unsigned long +DebutImage() : unsigned long +TailleImage() : unsigned long +TailleEntete() : unsigned long +Format() : unsigned long +ResolutionLargeur() : unsigned long +ResolutionHauteur() : unsigned long +Charger(nomFichier : string) : void +Sauvegarder(nomFichier : string) : void +Tourner90Droite() : void +Tourner90Gauche() : void +Tourner180() : void +Negatif() : void +RetournerHorizontal() : void +RetournerVertical() : void +NiveauDeGris() : void +Eclaircir(niveau : int) : void +Assombrir(niveau : int) : void +SeuillerNoirBlanc(niveau : int) : void +Colorier(coord : Coordonnee, l : int, h : int, couleur : Pixel) : void +Detourer(largeurGauche : int, largeurDroite : int, largeurHaut : int, largeurBas : int, couleur : Pixel) : void +Recadrer(coord : Coordonnee, l : int, h : int) : void +RechercherZone(l : int, h : int, couleur : Pixel) : Coordonnee +DessinerCroix(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel) : void +DessinerCarre(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel) : void +Dessiner7Segments(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel, message : string) : void +Dessiner11Segments(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel, message : string) : void +Dessiner7Segments(x : int, y : int, taille : int, epaisseur : int, couleur : int, message : string) : void +Dessiner11Segments(x : int, y : int, taille : int, epaisseur : int, couleur : int, message : string) : void +TracerDiagonale(debut : Coordonnee, fin : Coordonnee, epaisseur : int, couleur : Pixel) : void +TracerDiagonale(debutX : int, debutY : int, finX : int, finY : int, epaisseur : int, couleur : int) : void +TracerCercle(centre : Coordonnee, rayon : int, epaisseur : int, couleur : Pixel) : void +TracerCercle(centreX : int, centreY : int, rayon : int, epaisseur : int, couleur : int) : void +AfficherImageTexte(niveau : int) : void +AfficherImageTexte4Seuils() : void +AfficherImageTexte10caracteres(caracteres : char [0..*]) : void +AfficherTexteAvecImage(texte : string, pourcent : int) : void +AfficherTexteAvecImageFichier(nomFichierTxt : char [0..*], pourcent : int) : void +Histogrammes() : void +DissimulerTexte(texte : char [0..*]) : void +ExtraireTexte() : string +SeuillerNoirBlancMoyen() : int +SeuillagePourcentNoir(pourcent : int) : bool +DessinerCroixEtCoordonnees(coord : Coordonnee, tailleCroix : int, epCroix : int, tailleTexte : int, epTexte : int, coul : Pixel, droite : bool) : void +Dessiner36Segments(coord : Coordonnee, taille : int, epaisseur : int, couleur : Pixel, message : string) : void </pre>

Annexe 2 : bibliothèque QRcode qrcodegen.h

```

// QR Code generator library (C)
// Copyright (c) Project Nayuki. (MIT License)
// https://www.nayuki.io/page qr-code-generator-library
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
enum qrcodegen_Ecc {
    qrcodegen_Ecc_LOW = 0, // The QR Code can tolerate about 7% erroneous codewords
    qrcodegen_Ecc_MEDIUM, // The QR Code can tolerate about 15% erroneous codewords
    qrcodegen_Ecc_QUARTILE, // The QR Code can tolerate about 25% erroneous codewords
    qrcodegen_Ecc_HIGH, // The QR Code can tolerate about 30% erroneous codewords
};
enum qrcodegen_Mask {
    qrcodegen_Mask_AUTO = -1,
    qrcodegen_Mask_0 = 0,
    qrcodegen_Mask_1,
    qrcodegen_Mask_2,
    qrcodegen_Mask_3,
    qrcodegen_Mask_4,
    qrcodegen_Mask_5,
    qrcodegen_Mask_6,
    qrcodegen_Mask_7,
};
enum qrcodegen_Mode {
    qrcodegen_Mode_NUMERIC = 0x1,
    qrcodegen_Mode_ALPHANUMERIC = 0x2,
    qrcodegen_Mode_BYTE = 0x4,
    qrcodegen_Mode_KANJI = 0x8,
    qrcodegen_Mode_ECI = 0x7,
};
struct qrcodegen_Segment {
    enum qrcodegen_Mode mode;
    int numChars;
    uint8_t *data;
    int bitLength;
};
#define qrcodegen_VERSION_MIN 1 // The minimum version number supported in the QR Code Model 2 standard
#define qrcodegen_VERSION_MAX 40 // The maximum version number supported in the QR Code Model 2 standard
#define qrcodegen_BUFFER_LEN_FOR_VERSION(n) (((n) * 4 + 17) * ((n) * 4 + 17) + 7) / 8 + 1
#define qrcodegen_BUFFER_LEN_MAX qrcodegen_BUFFER_LEN_FOR_VERSION(qrcodegen_VERSION_MAX)
bool qrcodegen_encodeText(const char *text, uint8_t tempBuffer[], uint8_t qrcode[],
    enum qrcodegen_Ecc ecl, int minVersion, int maxVersion, enum qrcodegen_Mask mask, bool boostEcl);
bool qrcodegen_encodeBinary(uint8_t dataAndTemp[], int dataLen, uint8_t qrcode[],
    enum qrcodegen_Ecc ecl, int minVersion, int maxVersion, enum qrcodegen_Mask mask, bool boostEcl);
bool qrcodegen_encodeSegments(const struct qrcodegen_Segment segs[], int len,
    enum qrcodegen_Ecc ecl, uint8_t tempBuffer[], uint8_t qrcode[]);
bool qrcodegen_encodeSegmentsAdvanced(const struct qrcodegen_Segment segs[], int len, enum qrcodegen_Ecc ecl,
    int minVersion, int maxVersion, enum qrcodegen_Mask mask, bool boostEcl, uint8_t tempBuffer[], uint8_t qrcode[]);
bool qrcodegen_isAlphanumeric(const char *text);
bool qrcodegen_isNumeric(const char *text);
int qrcodegen_calcSegmentBufferSize(enum qrcodegen_Mode mode, int numChars);
struct qrcodegen_Segment qrcodegen_makeBytes(const uint8_t data[], int len, uint8_t buf[]);
struct qrcodegen_Segment qrcodegen_makeNumeric(const char *digits, uint8_t buf[]);
struct qrcodegen_Segment qrcodegen_makeAlphanumeric(const char *text, uint8_t buf[]);
struct qrcodegen_Segment qrcodegen_makeEci(long assignVal, uint8_t buf[]);
int qrcodegen_getSize(const uint8_t qrcode[]);
bool qrcodegen_getModule(const uint8_t qrcode[], int x, int y);

```