Module C/C++

Codage Base 64

Table des matières

Objectifs et ressources

Objectifs

Base64 est un codage de l'information utilisant 64 caractères, choisis pour être disponibles sur la majorité des systèmes.Il s'agit d'étudier ce codage, puis de programmer, en mode console, les fonctions permettant de coder des données binaires, quelle que soit leur taille, en base 64. Programmer les fonctions de décodage. Permettre le codage et le décodage de fichiers. 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`.

- Tableaux de caractères
- Traitement de bit
- Binaire et hexadécimal
- Langage C
- Algorithmie

Logiciels

- C++ Builder
- CodeBlocks

Annexe

- Table base 64

– Défi 1 – La table base64

Principe :

Un alphabet de 65 caractères est utilisé pour permettre la représentation de 6 bits par un caractère. Le 65e caractère (signe '=') n'est utilisé qu'en complément final dans le processus de codage d'un message. Ce processus de codage consiste à coder chaque groupe de 24 bits successifs de données par une chaîne de 4 caractères. On procède de gauche à droite, en concaténant 3 octets pour créer un seul groupement de 24 bits (8 bits par octet). Ils sont alors séparés en 4 nombres de seulement 6 bits (qui en binaire ne permettent que 64 combinaisons). Chacune des 4 valeurs est enfin représentée (codée) par un caractère de l'alphabet retenu. Ainsi 3 octets quelconques sont remplacés par 4 caractères.

Chaque valeur (chaque groupe de 6 bits) est utilisée comme index dans la table donnée ci-dessous. Le caractère correspondant est indiqué dans la colonne codage.

Valeur	Codage	Valeur	Codage	Valeur	Codage	Valeur	Codage
0	000000 A	17	010001 R	34	100010 i	51	110011 z
1	000001 B	18	010010 S	35	100011 j	52	110100 0
2	000010 C	19	010011 T	36	100100 k	53	110101 1
3	000011 D	20	010100 U	37	100101 l	54	110110 2
4	000100 E	21	010101 V	38	100110 m	55	110111 3
5	000101 F	22	010110 W	39	100111 n	56	111000 4
6	000110 G	23	010111 X	40	101000 o	57	111001 5
7	000111 H	24	011000 Y	41	101001 p	58	111010 6
8	001000 I	25	011001 Z	42	101010 q	59	111011 7
9	001001 J	26	011010 a	43	101011 r	60	111100 8
10	001010 K	27	011011 b	44	101100 s	61	111101 9
11	001011 L	28	011100 c	45	101101 t	62	111110 +
12	001100 M	29	011101 d	46	101110 u	63	111111 /
13	001101 N	30	011110 e	47	101111 v		
14	001110 O	31	011111 f	48	110000 w	(complément)	=
15	001111 P	32	100000 g	49	110001 x		
16	010000 Q	33	100001 h	50	110010 y		

L'intérêt de l'encodage base64 ne se trouve pas dans la représentation de données textuelles, mais dans la représentation de données binaires. Lorsque l'on veut représenter des données binaires (une image, un exécutable) dans un document textuel, tel qu'un courriel, la transcription hexadécimale en ASCII des octets multiplierait la taille par deux, l'encodage en base64 permet de limiter cette augmentation. Source : https://fr.wikipedia.org/wiki/Base64.

Exemple :

Soient les 3 octets : 0xFF 0xD8 0x26 (11111111 11011000 00100110).

En codage ASCII ces 3 octets deviennent : "FFD826" et utilisent 6 octets (un par caractère).

En codage Base64, les bits 11111111 11011000 00100110 sont regroupés par 6 bits : 111111 111101 100000 100110.

D'après la table de codage, nous obtenons "/9gm" seulement 4 octets sont utilisés.

? Donner le code correspondant à 0x 89 0x50 0x4E

? Décoder le code Rw0K

Création de la table de codage :

Nous déclarerons en variable globale le tableau de 64 caractères : **tableBase64**.

La fonction **InitialiserTable** permet de remplir le tableau **tableBase64** avec les caractères **'A','B','C'... ,'a','b','c'...'0','1','2'...,'+','/'**.

Il est possible d'utiliser une boucle avec des tests ou bien 3 boucles successives.

? Donner le code de la fonction InitialiserTable.

//Pour afficher le code binaire, on donne :

```
void AfficherBinaire(int entier, int nbBits)
{  for(int i=0;i<nbBits;i++) cout<<((entier>>(nbBits-i-1))&0x01);
}
```

? Donner le code de la fonction AfficherTable permettant d'afficher tableBase64 en précédant chaque caractère de son indice en décimal et en binaire 6bits.

⌨ Coder les fonctions ci-dessus ainsi que le programme principal visant à les tester.

//exemple de sortie console :

0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

– Défi 2 – Codage

Codage d'un groupe de 3 octets :

Il nous faut coder la fonction **void Code3octetsBase64(unsigned char* donnee, char* code)** permettant de coder en base 64 les 3 octets contenus dans le tableau **donnee**. Les 3 caractères seront stockés dans le tableau **code** et seront suivis du caractère de fin de chaîne (**0**).

Dans cette fonction, il faut déclarer un entier **nb24bits**. Puis stocker, dans cet entier, les 3 octets du tableau **donnee** (c'est à dire **donnee[0]** décalé de **16 bits** plus **donnee[1]** décalé de **8 bits** plus **donnee[2]**. Attention, mettre des parenthèses car l'opérateur d'addition est prioritaire sur les opérateurs de décalage.

//dans notre exemple :

```
donne[0]=0xFF  soit 11111111
donne[1]=0xD8  soit 11011000
donne[2]=0x26  soit 00100110
 11111111 00000000 00000000
+         11011000 00000000
+                  00100110
=11111111 11011000 00100110   stocké dans nb24bits
```

Pour les tests, il est préférable d'afficher le contenu binaire du **nb24bits** en appelant **AfficherBinaire**. Dans l'élément d'indice **4** du tableau code, il faut placer le caractère de fin de chaîne. Puis stocker dans un entier **i** les **6 bits** de poids faibles de **nb24bits** : il est nécessaire d'utiliser un masque en ET avec la valeur **0x3f**.

//dans notre exemple

```
  11111111 11011000 00100110
& 00000000 00000000 00111111  (0x3f)
= 00000000 00000000 00100110  stocké dans i
```

Dans l'élément d'indice **3** du tableau **code**, il faut placer l'élément d'indice **i** du tableau **tableBase64**. Décaler ensuite à droite **nb24bits** de **6 bits** afin de traiter le groupe suivant (**>>=**). Stocker à nouveau les **6 bits** de poids faible de **nb24bits** dans **i**.

Il faut renouveler ce traitement pour les éléments **2**, **1** et **0** du tableau **code**.

Donner le code de Code3octetsBase64.

Coder Code3octetsBase64. Puis, dans le programme principal, déclarer un tableau donnee de 3 octets initialisé à {0xff,0xd8,0x26}. Déclarer un tableau code de 5 caractères. Appeler Code3octetsBase64 puis afficher code.

//sortie console :

```
111111111101100000100110
/9gm
```

– Défi 3 – Décodage

IndiceCaractereBase64 :

Il est nécessaire de coder la fonction **unsigned char IndiceCaractereBase64(char caractere)** permettant de trouver l'indice de la table base 64 correspondant à un caractère donné. la fonction retourne la valeur de cet indice. par exemple **IndiceCaractereBase64('v')** retourne le nombre **47**.

Il y a 5 cas différents :	Si le caractère est compris entre **'A'** et **'Z'**	l'indice est **caractere-'A'**
	Si le caractère est compris entre **'a'** et **'z'**	l'indice est **caractere-'z'+26**
	Si le caractère est compris entre **'0'** et **'9'**	l'indice est **caractere-'0'+52**
	Si le caractère est **'+'**	l'indice est **62**
	Si le caractère est **'/'**	l'indice est **63**
	Dans les autres cas, le caractère n'est pas base64 :	retourner **64**.

Donner le code de IndiceCaractereBase64.

Coder IndiceCaractereBase64, puis tester cette fonction dans le programme principal.

Decode4caracteresBase64 :

Il nous faut coder la fonction **void Decode4caracteresBase64(unsigned char* donnee, char* code)** permettant de décoder dans le tableau **donnee** de 3 octets les 4 caractères du tableau **code**. Dans cette fonction, un entier nb24bits doit être déclarer. Stocker dans cet entier l'indice correspondant au premier caractère (en utilisant **IndiceCaractereBase64**) décalé de **18**, plus l'indice du 2ème décalé de **12**, plus l'indice du 3ème décalé de **6**, plus l'indice du 4ème. Attention, mettre des parenthèses car l'opérateur d'addition est prioritaire sur les opérateurs de décalage.

//dans notre exemple

```
IndiceCaractereBase64('/') vaut 63 soit 111111
IndiceCaractereBase64('9') vaut 61 soit 111101
IndiceCaractereBase64('g') vaut 32 soit 100000
IndiceCaractereBase64('m') vaut 38 soit 100110
 111111 000000 000000 000000
+       111101 000000 000000
+              100000 000000
+                     100110
=111111 111101 100000 100110  stocké dans nb24bits
```

Pour les tests, il est préférable d'afficher le contenu binaire du **nb24bits** en appelant **AfficherBinaire**. Stocker ensuite dans **donnee[2]** les **8 bits** de poids faible de **nb24bits** (masque en ET avec **0xff**), puis décaler **nb24bits** de **8 bits** à droite (**>>=**). Il faut renouveler l'opération pour **donnee[1]** puis **donnee[0]**.

Donner le code de Decode4caracteresBase64.

Coder Decode4caracteresBase64. Dans le main, Appeler Decode4caracteresBase64 puis afficher les 3 octets en hexadécimal.

//code d'affichage :

```
cout<<hex<<(int)donnee[0]<<" "<<(int)donnee[1]<<" "<<(int)donnee[2];
```

Où **hex** permet de passer en mode hexadécimal, et **(int)** permet de forcer l'affichage d'un nombre (et non d'un caractère).

//sortie console :

```
111111111101100000100110
ff d8 26
```

– Défi 4 – Codage d'un buffer de taille quelconque

Caractère complémentaire :

Un traitement spécial est effectué si moins de 24 bits sont disponibles à la fin de la séquence de données à coder (elle n'a pas forcément une taille multiple de 24 bits). Dans un tel cas, des zéros sont ajoutés à la droite des données initiales pour aller vers le multiple de 6 bits le plus proche. Chaque paquet de 6 bits est converti dans l'alphabet. Puis on ajoute des caractères '=' complémentaires pour former quand même 4 caractères.

Puisque les données d'entrée doivent être constituées d'un nombre entier d'octets, seuls trois cas sont possibles en fin de séquence :

- Il reste exactement 3 octets à coder (24 bits), alors on obtient directement 4 caractères sans traitement complémentaire ;
- Il reste seulement 2 octets (16 bits) à coder, alors on ajoute à droite 2 bits à zéros pour former 3 caractères de l'alphabet ($3\times6 = 16+2 = 18$ bits) suivis d'un quatrième caractère '=' en complément ;
- Il reste un seul octet (8 bits) à coder, alors on ajoute à droite 4 bits à zéros pour former 2 caractères de l'alphabet ($2\times6 = 8+4 = 12$ bits) suivis de deux caractères '=' en complément.

Source : https://fr.wikipedia.org/wiki/Base64. Pour tester vos codes : https://cryptii.com/pipes/base64-to-binary.

CodeNoctetsBase64 :

Il nous faut coder la fonction **void CodeNoctetsBase64(unsigned char* donnee, int nbOctetsDonnees, char* code)**. Cette fonction codera en base64 **nbOctetsDonnees** octets du tableau **donnee**.

Dans le code de cette fonction, un entier **resteEnFinDeSequence** est créé, il contiendra le nombre de bits restants à la fin de la séquence. Selon **resteEnFinDeSequence** il faudra ajouter éventuellement un ou bien deux '=' dans le code base64. Cet entier stockera le reste de la division de **nbOctetsDonnees*8** par **24** (utiliser l'opérateur **%** -modulo-). Afficher cet entier pour les tests.

Si **resteEnFinDeSequence** vaut **16**

Mettre un **0** à dans l'élément d'indice **nbOctetsDonnees** du tableau **donnee**, incrémenter alors **nbOctetsDonnees**.

Si **resteEnFinDeSequence** vaut **8**

Faire 2 fois le traitement précédent.

Nous avons besoin de déclarer un entier **j** initialisé à **0** pour servir d'indice dans le tableau **code**.

Pour **i** allant de **0** à **nbOctetsDonnees** par valeur inférieure, i allant de 3 en 3 (**i+=3**)

Appeler **Code3octetsBase64** en passant en argument **donnee+i** et **code+j**, puis ajouter **4** à **j** en fin de boucle.

Ajouter i à donnee (c'est l'adresse du tableau) permet de se déplacer dans le tableau donnee de i en i. Idem pour code+j.

Il reste à ajouter les caractères '=' lorsque le nombre de bits de la donnée n'est pas un multiple de 24.

Sachant que le dernier caractère de la chaîne **code** porte l'indice **strlen(code)-1**, que l'avant dernier porte l'indice **strlen(code)-2** :

Si **resteEnFinDeSequence** vaut **16**

Placer un '=' à la place du dernier caractère de **code**.

Si **resteEnFinDeSequence** vaut **8**

Placer un '=' à la place de chacun des 2 derniers caractères de **code**.

Donner le code de CodeNoctetsBase64.

Coder CodeNoctetsBase64. Dans le programme principal, passer le tableau donnee à une taille maximum de 1500, et le tableau code à une taille de 2001. Déclarer un entier nbOctets qui sera initialisé alternativement à 6 puis 5 puis 4 pour les tests. Appeler la fonction CodeNoctetsBase64.

//exemples de sortie console :

unsigned char donnee[1500]={0xff,0xd8,0x26,0xb0,0x41,0x78}; int nbOctets=6;

```
reste en fin de sequence :0 bits
111111111101100000100110
101100000100000101111000
/9gmsEF4
```

unsigned char donnee[1500]={0xff,0xd8,0x26,0xb0,0x41}; int nbOctets=5;

```
reste en fin de sequence :16 bits
111111111101100000100110
101100000100000100000000
/9gmsEE=
```

unsigned char donnee[1500]={0xff,0xd8,0x26,0xb0}; int nbOctets=4;

```
reste en fin de sequence :8 bits
111111111101100000100110
101100000000000000000000
/9gmsA==
```

– Défi 5 – Décodage d'un buffer de taille quelconque

Il ne reste plus qu'à coder la fonction **int DecodeNoctetsBase64(unsigned char* donnee, char* code)**. Cette fonction décode dans **donnee** le code base64 passé en argument, elle retourne le nombre d'octets de donnée.

Dans cette fonction, nous utiliserons l'indice **j** pour le tableau **donnee** et l'indice **i** pour la tableau **code**. Déclarer **j** et l'initialiser à **0**.

Pour **i** allant de **0** à la longueur de code (**strlen**) par saut de **4**,

Appeler la fonction **Decode4octetsBase64** en passant en argument **donnee+j** et **code+i**.

En fin de boucle, ajouter **3** à l'indice **j**.

Si le dernier caractère de **code** est **'='**, décrémenter **j**.

Si l'avant-dernier caractère de **code** est **'='**, décrémenter à nouveau **j**.

L'indice **j** contient donc le nombre d'octets de donnée : il faut le retourner.

? Donner le code de DecodeNoctetsBase64.

⌨ Coder DecodeNoctetsBase64. Dans le programme principal, initialiser le tableau code puis appeler la fonction DecodeNoctetsBase64, afficher sa valeur de retour. Afficher en hexadécimal toutes les données en utilisant une boucle.

//exemples de sortie console :

char code[2001]="/9gmsA=="

```
11111111110110000100110
10110000000100000100000
4 octets de donnee
ff d8 26 b0
```

char code[2001]="/9gmsEE="

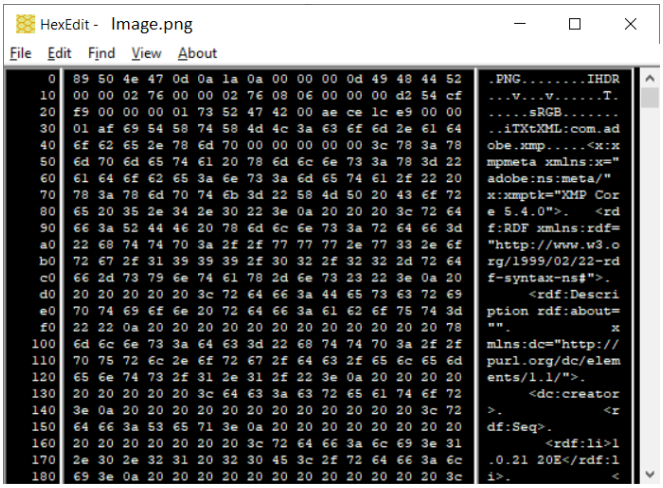
```
11111111110110000100110
10110000010000010100000
5 octets de donnee
ff d8 26 b0 41
```

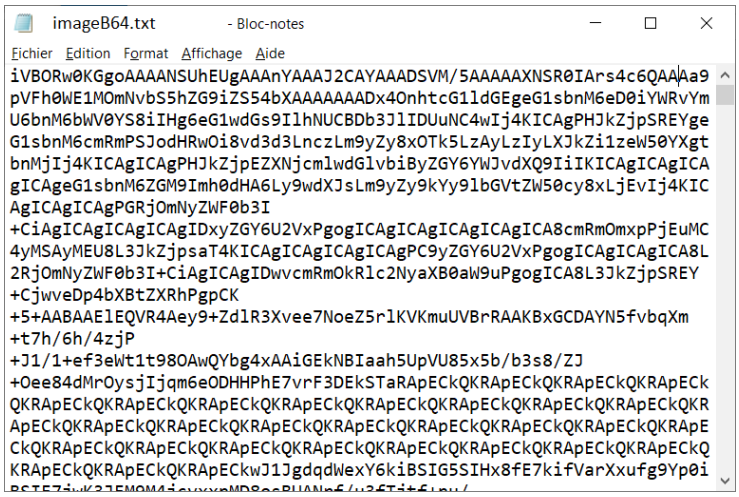
char code[2001]="/9gmsEF4"

```
11111111110110000100110
10110000010000010111000
6 octets de donnee
ff d8 26 b0 41 78
```

– Défi 6 – Codage et Décodage de fichiers

En C++ : utiliser les classes **ifstream** et **ofstream**.
En C : utiliser **FILE***





Codage :

Programmer en mode console, le programme permettant de coder un fichier binaire en base64 et de générer un fichier texte contenant le code.

Donner le code du programme.

Coder et tester le programme en mode console.

//exemple de sortie console :

```
Entrer le nom du fichier a coder en base64 : image.png
Entrer le nom du fichier resultat : imageB64.txt
Resultat du codage :
Binaire : 203 072 octets
Base 64 : 270 764 octets
```

Décodage :

Programmer également le décodage d'un fichier texte base64 pour générer le fichier binaire.

Donner le code du programme.

Coder et tester le programme en mode console.

//exemple de sortie console :

```
Entrer le nom du fichier base64 a decoder : imageB64.txt
Entrer le nom du fichier resultat : image.png
Resultat du decodage :
Base 64 : 270 764 octets
Binaire : 203 072 octets
```

Annexe : Table base 64

Valeur	Codage	Valeur	Codage	Valeur	Codage	Valeur	Codage
0	000000 A	17	010001 R	34	100010 i	51	110011 z
1	000001 B	18	010010 S	35	100011 j	52	110100 0
2	000010 C	19	010011 T	36	100100 k	53	110101 1
3	000011 D	20	010100 U	37	100101 l	54	110110 2
4	000100 E	21	010101 V	38	100110 m	55	110111 3
5	000101 F	22	010110 W	39	100111 n	56	111000 4
6	000110 G	23	010111 X	40	101000 o	57	111001 5
7	000111 H	24	011000 Y	41	101001 p	58	111010 6
8	001000 I	25	011001 Z	42	101010 q	59	111011 7
9	001001 J	26	011010 a	43	101011 r	60	111100 8
10	001010 K	27	011011 b	44	101100 s	61	111101 9
11	001011 L	28	011100 c	45	101101 t	62	111110 +
12	001100 M	29	011101 d	46	101110 u	63	111111 /
13	001101 N	30	011110 e	47	101111 v		
14	001110 O	31	011111 f	48	110000 w	(complément)	=
15	001111 P	32	100000 g	49	110001 x		
16	010000 Q	33	100001 h	50	110010 y		