A C++ Base64 Coder/Decoder Class

Tom Wheeler, NØGSG
twheeler@kc.devry.edu

Introduction

Often binary data must be transmitted on a communications channel that can only handle ASCII or text information. Binary data can have any value, but text channels can only handle printable character values and a small set of control characters (such as the carriage return). The traditional solution is to program the sender to encode the binary data into text characters which will then be interpreted at the receiver as binary data. Various schemes have been utilized over the years to accomplish this.

Hexadecimal or Base16 coding and its variants (such as Intel hex) are still used to upload machine-language data directly to microprocessor and microcontrollers. Each transmitted character represents a group of four binary bits. For example, the group

```
:10010000214601360121470136007EFE09D2190140
```

is the Intel Hex representation of the 16-byte (10_{16} octet) data block {21460136012147013600 7EFE09D21901} starting at memory address 0100_{16} and having an 8-bit checksum of 40_{16}. The ':' character is used for synchronization; it marks the beginning of the data record.

Base16 coding is inherently inefficient - at best, it sends 8 bits to represent each 4-bit group; only 50% or less of the channel capacity is utilized. Base32 coding and Base64 coding are more efficient adaptations of this same idea.

Base64 coding is extensively used in Internet and Web communications. It allows pure binary data to be sent over a "text only" channel. The "text only" channel is pretty much a modern relic; modern transport (OSI Layer IV) protocols such as TCP can easily handle binary data. The problem is that application protocols (OSI Layer VII+) such as HTTP (used for web transactions), SMTP (used for transferring e-mail), and others just weren't designed with binary data in mind. Base64 encoding allows these text-only protocols to handle any type of data.

The Base64 system encodes groups of six data bits into each transmitted character symbol. At best, the efficiency is about 6/8 or 75%, which although better than the 50% figure for Base16, still leaves a bit to be desired. Table 1 shows the Base64 code set.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
10	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
20	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
30	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Table 1: Base 64 Code Set (Based on RFC3548; RFC4648)

Encoding six bits at a time also introduces some clumsiness into the coding process, because transmitted bytes can *span* symbols. For example, the three character message "*Man*" (in ASCII) contains the code sequence { 0x4D, 0x61, 0x6E }. (The prefix "0x" indicates a hexadecimal or base 16 number). This message encodes quite easily:

Text Character	'M'		'a'		'n'	
Hex ASCII Code	4D		61		6E	
Binary ASCII Code	0 1 0 0 1 1 0 1		0 1 1 0 0 0 0 1		0 1 1 0 1 1 1 0	
Base64 Grouping	0 1 0 0 1 1	0 1 0 1 1 0	0 0 0 1 0 1	1 0 1 1 1 0		
Base64 Symbol	'T'	'W'	'F'	'u'		

Table 2: Encoding the Text "Man" (Adapted from http://en.wikipedia.org/wiki/Base64)

This message encoded neatly because the number of bits within, 24, also happened to be an integer multiple of 6 (the quantum of Base64 encoding). Note that encoding utilizes *spanning*. A Base64 symbol maps to portions of two adjacent input bytes; this introduces *positional sensitivity* -- the same byte sequence may have different Base64 representations, based on its offset within the data stream. What if an extra byte is added to the message? Table 3 shows how this is handled.

Text Character	'M'		'a'		'n'		'e'			
Hex ASCII Code	4D		61		6E		65			
Binary ASCII Code	01001101		01100001		01101110		01100101			
Base64 Grouping	010011	010110	000101	101110			011001	010000	PAD	PAD
Base64 Symbol	'T'	'W'	'F'	'u'			'Z'	'Q'	'='	'='

Table 3: Encoding Data with Padding

The addition of the fourth character complicated the encoding considerably! The message now has 30 bits (4 bytes at 8 bits/byte), and 30 isn't an integer multiple of the six-bit packets used by the encoder. To solve the problem, Base64 always encodes the message with a multiple of four symbols. The symbol sequence 'ZQ' encodes the final byte for the letter 'e'. Note that the 'Q' actually encodes only *two* of the bits in this final byte -- the remaining zero bits in 'Q' are *padding*.

The '=' symbols at stream end are also padding -- they represent binary zeros to be appended to the recovered data, if needed. In this example, a single '=' would theoretically be an adequate way of padding and terminating the message, but the rules for Base64 encoding require that all messages contain an integer multiple of four symbols, so the second '=' is a requirement, even though it is redundant.

Programmers have been known to pull hair out implementing coder/decoders (codecs) for systems like this. If you feel that this is unnecessarily complicated, you're probably right!

A Codec Class for Base64

The `Base64Codec` class is a simple and robust implementation of a Base64 coder and decoder. It contains two primary functional parts:

- An encoder. The encoder is passed a set of binary data; this data is converted into ASCII Base64 symbols. The encoder also contains measures to prevent buffer overrun.

- A decoder. The decoder tests the Base64 string for validity and decodes the string back into pure binary form.

Two Visual C++ 6.0 projects, "`Base64`" and "`Base64GUI`" are on this web site for you to download. They contain the source code and executable modules to demonstrate functionality of the `Base64Codec` class.

Simple Code Example: Coding and Decoding a Message

The following code is from the "Base64" project. It illustrates the simplicity of using the `Base64Codec` class.

```
char Base64buf[1024];  // A buffer to hold the Base-64 encoded string
int nResult;           // Result of the conversion (how many characters or bytes)
Base64Codec aCodec;    // Instantiate an instance of the codec
char BinBuf[1024];     // Buffer for the recovered binary data

char* testString;
testString = "Wherever you go, there you are.";      // Some ASCII data here


printf("Test string: \"%s\" (len=%d)\n\n", testString, strlen(testString) );

// You can ask the codec how big the Base64 string will be. This is quite handy and
// recommended if you plan on dynamically allocating buffer space.

printf("The codec estimates that %d bytes are needed to encode this string.\n",
                      aCodec.EstimateBase64EncodeSize( strlen(testString)) );

// Do the actual encode here
nResult = aCodec.Encode( Base64buf, 1023 , testString , strlen(testString) );
printf("\nResult Code: %2d characters in Base64 stream.\nOriginal Data: %-10s\nBase64
Stream: %-10s\n\n", nResult, testString, Base64buf );

// Then decode it back into binary (ASCII here)
nResult = aCodec.Decode(BinBuf,1023,Base64buf);
printf("\n\nDecode: %2d bytes of data\n\n", nResult );
// Since we decoded as a string and didn't include the '\0' in the encode, put it on
// the string here. (Alternately, could have passed (strlen(testString)+1) above.
BinBuf[nResult]=0;
printf("Decode as string: %s\n", BinBuf );
```

Benchmarks and Testing

The `Base64Codec` class has been benchmarked and tested on a 2 GHz class Windows 2000 desktop system. The testing process was as follows:

- A group of 100 JPEG binary files varying in size from 100 kB to 40 MB each were fed into the encoder, and output as text files.
- The resulting Base64 text files were then decoded back into binary and compared with the originals using the *Windiff* file comparison utility provided with Visual C++.

Even though the codec is a first pass attempt, it is very fast and robust. Coding a 40 MB JPEG data file and writing the Base64 data to disk required less than two seconds. Smaller files encoded *much* faster. The decoding process was of similar speed. No data errors were observed; all recovered binary files were identical to the original JPEG data (with the exception of the time stamps).

Summary

It's not hard to accomplish Base64 encoding and decoding within your own C++ programs. The `Base64Codec` class is a simple and efficient solution that is much better than pulling out your hair!