How to Base64

by Randy Charles Morin

Continuing in the line of Internet Protocol and specifically, how to send emails. I've been promising you guys that I'd tell you how to send email attachments. One problem, I first have to tell you how to encode binary files in the base 64 data representation format.

You see, when text files are attached to SMTP emails, the text files can be attached in their plain text format. But binary files cannot be attached without some form of encoding. The encoding used in SMTP and many other Internet Protocols is base 64.

Another interesting thing about base 64 is that it is one of the only data representation schemes in modern use that doesn't encrypt or compress the data. In just represents data in a different form. This is also true of another data representation protocol that is becoming very important in modern computing, SOAP. This discussion isn't about SOAP, so I'll leave it in the dust for now.

The problem with binary attachments is that they don't read very well when encapsulated in other documents. The format of a binary file is basically a length followed by a series of bytes. The bytes may contain zeros, which in some documents represent end of files. The bytes might also contain series of bytes that have other special representations that may skew the encapsulation of the attachment.

The solution to this problem was to create a data representation protocol that would allow binary data to be encapsulated within another document. The defacto standard for doing this is called base 64. In fact, some applications use base 64 as an encryption technique, albeit a weak one. Basic authentication over the Internet encrypts the username-password using base 64.

Encoding

Base 64 data representation is based on a 64-character alphabet. The alphabet is shown in Table 1.

Table 1: Base64 Alphabet

Sequence	Characters
0 … 25	'A' … 'Z'
26 … 51	'a' … 'z'
52 … 61	'0' … '9'
62	'+'
63	'/'

A binary file is a series of zero and ones. If you grouped those zeros and ones in sets of 6, then you would get a number between 0 and 63 for each set. Converting them using the base64 alphabet allows you to convert binary zeros and ones into a compressed and human readable format. By compressed, I mean that you can represent six bits with one character. This is six times less than representing each bit with a '0' or '1' character.

Let me present you with a simple example of converting a small binary string into base 64 notation.

```
'001100110011'
```

This 12-bit binary string can be divided into two sets of 6 bits.

```
'001100' & '110011'
```

If you convert the 6-bit binary number to base-10 notation, the notation understood by you and me, then you get 12 and 51.

```
0 × 32 + 0  × 16 + 1 × 8 + 1 × 4 + 0 × 2 + 0 × 1 = 12
1 × 32 + 1  × 16 + 0 × 8 + 0 × 4 + 1 × 2 + 1 × 1 = 51
```

Now if you look up 12 and 51 in the base-64 alphabet, they are 'M' for 12 and 'z' for 51. So we can effectively represent this binary sequence in two human readable characters.

'Mz'

Most computers group binary files in sets of 8 bits, called a byte. The problem with this grouping is that it cannot be represented in one character and also be human readable. Often bytes are represented in the hexadecimal data representation format (or set of 4-bits) that is maybe a little more human readable than base-64, but also slightly less space efficient.

The binary string '000111000111' would be '1C7' in hexadecimal notation, or three hexadecimal characters. The equivalent base-64 notation is 'HH', only two base-64 characters. So, as you can see, base-64 has a special efficiency (compression) advantage over the more popular hexadecimal notation.

Padding

The base-64 data representation method does have one failing, not present in the hexadecimal notation. Hexadecimal can efficiently represent a byte using exactly two hexadecimal characters, whereas base-64 cannot efficiently represent one byte of data. Since computers generally organize data into sequences of bytes, the base-64 requires some additional rules when a sequence of bytes cannot be efficiently represented. Groups of 3 bytes can efficiently be represented by reorganizing the bytes into 4 base-64 characters.

Let's take a sample 3 bytes sequence to demonstrate this fact.

```
'01010101' '00100100' '00010001'
```

In hex this number would be '552411'. If you regroup these 3 bytes into series of 6-bits, then you get exactly four groups of 6-bits.

```
'010101' '010010' '010000' '010001'
```

In base-64 this representation is 'VSQR'. Again, this is perfectly efficient, because I groups 3 bytes into 4 base-64 characters. So, whenever a binary sequence is exactly divisible by 3 bytes, then the representation in base-64 is efficient.

In the case where a binary sequence is not an integral size of 3 bytes, then you have a representation problem. There are two cases to consider. When after regrouping each series of 3 bytes as 4 base-64 characters, you will have zero, one or two bytes left over. The problem occurs in the cases where you have one or two bytes remaining. How do you represent these instances? Let's consider each case separately.

The first case where you have one byte remaining, you should pad two additional bytes with all zeros onto the end of the binary sequence. You can then represent the one byte with two base-64 characters followed by two padding characters. The padding character in base-64 is '='.

Let consider an example.

```
'00000001'
```

Pad the single-byte instance with two more bytes of zeros.

```
'00000001' '00000000' '00000000'
```

Now break up the binary sequence in sets of six bytes.

```
'000000' '010000' '000000' '000000'
```

Take the first two base-64 characters and pad two '=' characters to the end of the sequence.

```
'AQ=='
```

The second case is where you have two bytes remain.

```
'00000010' '00000001'
```

Here you should pad one additional zero byte to the end of the binary sequence.

```
'00000010' '00000001' '00000000'
```

Now break up the binary sequence in sets of six bytes.

```
'000000' '100000' '000100' '000000'
```

We then take three base-64 characters and pad with one '=' sign.

```
'AgE='
```

Line Length

One last issue in encoding a byte stream is limiting the length of any line without the base-64 stream. The stream would not be human readable if it was one line several thousand characters long. In order to improve human readability in the stream the base64 specification requires that each line be at most 76 base-64 characters in length. After each 76 base-64 characters, I insert a carriage return and line feed into the stream. This increases the stream length by 3%, which is acceptable.

Decoding

Some special processing must be used in decoding the stream. First you group every 4 base-64 characters together and realign as 3 bytes. This similar to the algorithm I introduced when encoding so I won't go into any detail.

Two cases must be considered in the terminating bytes of the stream. If the last characters of the base-64 stream, you may encounter one or two padding characters, the equal character (=). If you encounter one pad character in the last four characters, then you should drop the trailing byte in the sequence. If you encounter two pad characters in the last four characters, then you should drop the two trailing bytes in the sequence. This is the reverse of the padding during the encoding algorithm.

Another important thing to do is to ignore none base-64 characters in the stream. During encoding I dropped carriage returns and line feeds into the stream to break up the lines. It is also allowed to drop other non base-64 characters into the stream. For these reasons, I scan and remove non base-64 characters from the stream before decoding.

Code

The interface to the class is actually pretty simple.

```
class Base64
{
        static char Encode(unsigned char uc);
        static unsigned char Decode(char c);
        static bool IsBase64(char c);

public:
        static std::string Encode(
                const std::vector<unsigned char> & vby);
        static std::vector<unsigned char> Decode(
                const std::string & str);
};
```

I have two public static methods, one for encoding and one for decoding streams. I also created four private helper functions that are used in the implementation. Let me start by explaining the four helper-functions.

The first is the Encode function, which translates one six-bit pattern into a base-64 character. As there is not efficient six-bit data structure in C++, I used the unsigned char to represent a six-bit stream of data. The higher two bits in this case will always be zero.

```
inline char Base64::Encode(unsigned char uc)
{
        if (uc < 26)
        {
                return 'A'+uc;
        }

        if (uc < 52)
        {
                return 'a'+(uc-26);
        }

        if (uc < 62)
        {
                return '0'+(uc-52);
        }

        if (uc == 62)
        {
                return '+';
        }

        return '/';
};
```

I could also have used a switch statement, but I thought this was more readable and depending on the compiler, it might also be optimized.

The next function translates one base-64 character into a six-bit pattern.

```
inline unsigned char Base64::Decode(char c)
{
	if (c >= 'A' && c <= 'Z')
	{
		return c - 'A';
	}

	if (c >= 'a' && c <= 'z')
	{
		return c - 'a' + 26;
	}

	if (c >= '0' && c <= '9')
	{
		return c - '0' + 52;
	}

	if (c == '+')
	{
		return 62;
	};

	return 63;
};
```

Again, I could have used a switch statement, but chose not to.

The last helper function returns true is a character is a valid base-64 character and false otherwise.

```
inline bool Base64::IsBase64(char c)
{
	if (c >= 'A' && c <= 'Z')
	{
		return true;
	}

	if (c >= 'a' && c <= 'z')
	{
		return true;
	}

	if (c >= '0' && c <= '9')
	{
		return true;
	}

	if (c == '+')
	{
		return true;
	};

	if (c == '/')
	{
		return true;
	};

	if (c == '=')
	{
		return true;
	};

	return false;
};
```

The Encode static method takes an array of 8-bit values and returns a base-64 stream.

```
inline std::string Base64::Encode(
	const std::vector<unsigned char> & vby)
{
	std::string retval;

	if (vby.size() == 0)
	{
		return retval;
	};

	for (int i=0;i<vby.size();i+=3)
	{
		unsigned char by1=0,by2=0,by3=0;
		by1 = vby[i];
		if (i+1<vby.size())
		{
			by2 = vby[i+1];
		};
		if (i+2<vby.size())
		{
			by3 = vby[i+2];
		}

		unsigned char by4=0,by5=0,by6=0,by7=0;
		by4 = by1>>2;
		by5 = ((by1&0x3)<<4)|(by2>>4);
		by6 = ((by2&0xf)<<2)|(by3>>6);
		by7 = by3&0x3f;

		retval += Encode(by4);
		retval += Encode(by5);

		if (i+1<vby.size())
		{
			retval += Encode(by6);
		}
		else
		{
			retval += "=";
		};

		if (i+2<vby.size())
		{
			retval += Encode(by7);
		}
		else
		{
			retval += "=";
		};

		if (i % (76/4*3) == 0)
		{
			retval += "\r\n";
		}
	};

	return retval;
};
```

I checked for the zero length. The algorithm would also work for zero length input stream, but I'm pretty adamant about handling border conditions. They are often the culprits of run-time production failures.

The algorithm goes thru each three bytes of data at a time. The first thing I do is to shift the bits around from three 8-bit values to four 6-bit values. Then I encode the 6-bit values and add then one at a time to the output stream. This is actually quite inefficient. The STL character array is being allocated one byte at a time. The algorithm would be much faster, if I pre-allocated that array. I'll leave that as an optimization practical exercise for the reader.

The Decode static method takes a base-64 stream and converts it to an array of 8-bit values.

```
inline std::vector<unsigned char> Base64::Decode(
        const std::string & _str)
{
        std::string str;
        for (int j=0;j<_str.length();j++)
        {
                if (IsBase64(_str[j]))
                {
                        str += _str[j];
                }
        }

        std::vector<unsigned char> retval;

        if (str.length() == 0)
        {
                return retval;
        }

        for (int i=0;i<str.length();i+=4)
        {
                char c1='A',c2='A',c3='A',c4='A';
                c1 = str[i];
                if (i+1<str.length())
                {
                        c2 = str[i+1];
                };
                if (i+2<str.length())
                {
                        c3 = str[i+2];
                };
                if (i+3<str.length())
                {
                        c4 = str[i+3];
                };

                unsigned char by1=0,by2=0,by3=0,by4=0;
                by1 = Decode(c1);
                by2 = Decode(c2);
                by3 = Decode(c3);
                by4 = Decode(c4);

                retval.push_back( (by1<<2)|(by2>>4) );

                if (c3 != '=')
                {
                        retval.push_back( ((by2&0xf)<<4)|(by3>>2) );
                }

                if (c4 != '=')
                {
                        retval.push_back( ((by3&0x3)<<6)|by4 );
                };
        };

        return retval;
};
```

The decoding algorithm handles four base-64 characters at a time. First the characters are decoded, and the bits are shifted around into three 8-bit values and inserted into the output array.

Sample

I thought I'd also provide sample code for using the class.

```
// base64.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "base64.h"
#include <iostream>
#include <fstream>

void inout(char * szFilename);

int main(int argc, char* argv[])
{
        if (argc != 2)
        {
                std::cerr << "Usage: base64 [file]";
        }
        inout(argv[1]);
        return 0;
};

void inout(char * szFilename)
{
        std::vector<unsigned char> vby1;

        {
                std::ifstream infile;
                infile.open(szFilename, std::ios::in+std::ios::binary);
                if (!infile.is_open())
                {
                        std::cerr << "File not open!";
                        return;
                };
                infile >> std::noskipws;
                unsigned char uc;
                while (true)
                {
                        infile >> uc;
                        if (!infile.eof())
                        {
                                vby1.push_back(uc);
                        }
                        else
                        {
                                break;
                        }
                }
                std::cout << "File length is " << vby1.size() << std::endl;
        };

        std::vector<unsigned char>::iterator j;

        std::string str = kbcafe::Base64::Encode(vby1);

        std::cout << "Interim" << std::endl;
        std::cout << str << std::endl << std::endl;

        std::vector<unsigned char> vby2;
        vby2 = kbcafe::Base64::Decode(str);

        std::vector<unsigned char>::iterator k;
        int i=1;
        j = vby2.begin();
        k = vby1.begin();

        if (vby1.size() != vby2.size())
        {
                std::cerr << "Error in size " << vby1.size() << " "
                        << vby2.size() << std::endl;
        }

        for (;j!=vby2.end();j++,k++,i++)
        {
                if (*j != *k)
                {
                        std::cerr << "Error in translation " << i
                                << std::endl;
                }
        };

        {
                std::ofstream outfile;
                std::string strOutfile = szFilename;
                strOutfile += ".bak";
                outfile.open(strOutfile.c_str(),
                        std::ios::out+std::ios::binary);
                if (!outfile.is_open())
                {
                        std::cerr << "File not open!";
                        return;
                };
                unsigned char uc;
                j = vby2.begin();
                for (;j!=vby2.end();j++)
                {
                        uc = *j;
                        outfile << uc;
                }
        };

        return;
}
```

This code reads a file from disk into a byte array, and then converts the array to base-64, and then converts it back to binary. If also compares the input and output byte arrays for size and content differences and finally writes the output array to a file.

Conclusion

For more information about base 64, I suggest consulting the various RFCs. Base 64 is described in RFC 1521, http://www.ietf.org/rfc/rfc1521.txt?number=1521. You can download the base64 code from the KBCafe MSN community.

Author

Randy Charles Morin is the Lead Architect of SportMarkets Development from Toronto, Ontario, Canada and lives with his wife and two kids in Brampton, Ontario. He is the author of the www.kbcafe.com website, author of Wiley's Programming Windows Services book and co-author of many other programming books.