Laboratory 9: Bit Twiddling and Related Concepts

Introduction: There are many programming test questions that you could be asked to talk through in a technical interview, there are many sites that list these types of question. For example you could look at the archive list at Project Euler (pronounced “Oil-er”)

https://projecteuler.net/

When you look at these problems you begin to realise there are many ways of solving each of them. For example consider the following problem,

Part1: Write a method that detects if an integer passed to it is a power of two, let’s have a go....

Launch VS2010 and create a Windows 32 bit console application called *Bits*.

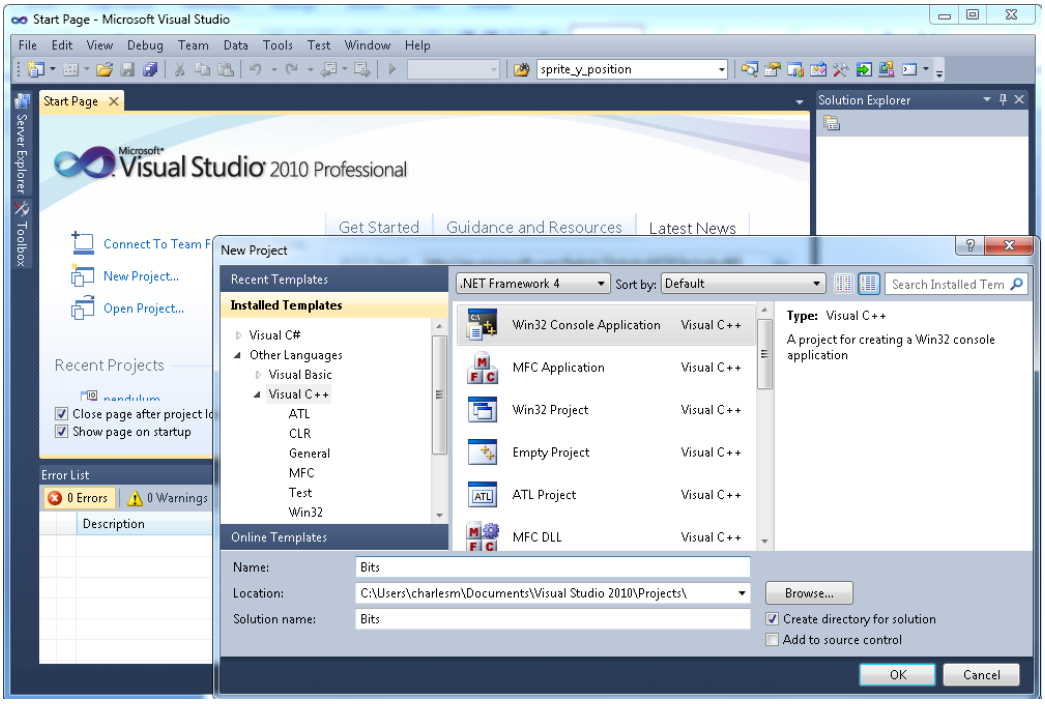

Figure 1 Creating a simple console application in which to create some code.

Accept the messages and then edit the Bits.cpp file so that you can add a simple program to say if a number is a power of two or not.

Try 1: Perhaps the first approach you consider is to check the 32 possibilities. The code to do this is shown on the next page, cut and paste it into your project and get it up and running.

```
#include "stdafx.h"

bool power_of_two(unsigned int);

int _tmain(int argc, _TCHAR* argv[])
{
	unsigned int x;

	do{
		printf("Enter a number (0 to finish):");
		scanf(" %u",&x);

		if(power_of_two(x))
		{
			printf("%u is a power of two\n",x);
		}
		else
		{
			printf("%u is not a power of two\n",x);
		}

	}while(x!=0);

	return 0;
}

bool power_of_two(unsigned int x)
{
	bool flag=false;
	if(x==1) flag=true;
	if(x==2) flag=true;
	if(x==4) flag=true;
	if(x==8) flag=true;
	if(x==16) flag=true;
	if(x==32) flag=true;
	if(x==64) flag=true;
	if(x==128) flag=true;
	if(x==256) flag=true;
	if(x==512) flag=true;
	if(x==1024) flag=true;
	if(x==2048) flag=true;
	if(x==4096) flag=true;
	if(x==8192) flag=true;
	if(x==16384) flag=true;
	if(x==32768) flag=true;
	if(x==65536) flag=true;
	if(x==131072) flag=true;
	if(x==262144) flag=true;
	if(x==524288) flag=true;
	if(x==1048576) flag=true;
	if(x==2097152) flag=true;
	if(x==4194304) flag=true;
	if(x==8388608) flag=true;
	if(x==16777216) flag=true;
	if(x==33554432) flag=true;
	if(x==67108864) flag=true;
	if(x==134217728) flag=true;
	if(x==268435456) flag=true;
	if(x==536870912) flag=true;
	if(x==1073741824) flag=true;
	if(x==2147483648) flag=true;
	return(flag);
}
```

Listing 1 A “brute force” approach to detecting if a number is a power of two.

By this stage you should be saying that there must be a better way. In fact the above method is probably the easiest to understand and formally verify that it will work.

What do you mean by "better"? Is there a faster way? Is there a way that uses less lines of code? Is there a way that is less prone to typing errors? Is there a way that is easy to understand?

Try 2: So perhaps we could take a new approach, we know the logarithm to base 2 of the number should be an integer value if the number is a power of two.

$$Log_2(8) = 3 \qquad \text{or} \qquad 2*2*2 = 2^3 = 8$$

The following code uses this approach. Replace the existing method with the new method. You will need to add the maths library also.

```
...
#include <math.h>
...

bool power_of_two(unsigned int x)
{
	double l=log((double)x)/log(2.0);
	if (l==floor(l))
	{
	 return(true);
	}
	else
	{
	 return(false);
	}
}
```

Try 3: Using floating point has the disadvantage if requiring the addition of a large library and you also have in the back of your mind that there could be issues relating to round-off errors. The code is shorter but probably not a way forward.

If you think about it, a number that is a power of two only has one bit set. Counting the bits is known as a "popcount", short for population count. Using the shift operator (or dividing by 2) this is a relatively easy task to implement. See the code that follows and try it.

```
bool power_of_two(unsigned int x)
{
	int popcount=0;
	for(int n=0;n<32;n++)		// For 32 bits
	{
		if ((x&1)==1) popcount++; // Check bit 0
		x=x>>1;			// Shift x right by 1 bit
	}
	if(popcount==1) return(true); else return(false);
}
```

Try 4: The above approach is elegant but does require a loop and a running total so it is not likely to be very fast. A look around the web shows that there is a method for detecting if a number is a power of two. The method is based on an approach know as “bit twiddling”. See the following website for an interesting list of examples.

https://graphics.stanford.edu/~seander/bithacks.html

So let’s just try it....

```
bool power_of_two(unsigned int x)
{
	if(x&(x-1)) return(false); else return(true);
}
```

It nearly works perfectly, however it fails for the input of of x=0 where it says it is a power of two. There is a fix...

```
bool power_of_two(unsigned int x)
{
	if (x && !(x&(x-1))) return(true); else return(false);
}
```

This is looking really good, just a few logical operations and we have the answer. The big problem is that it is really difficult to see how it works. So let’s have a look...

Note:

& means AND the individual bits together in the two numbers.

&& means logical AND.

Integer values equal to 0 are considered FALSE and all integer values not equal to zero (1,2,3...etc) are considered TRUE.

Analysis: How can x&(x-1) detect if a number is a power of two?

Lets choose a number, say x=1000, and limit the analysis to 16 bits.

```
X   = 1000 = 0000,0011,1110,1000b
&
X-1 = 999 = 0000,0011,1110,0111b
            0000,0011,1110,0000b
```

Interesting but I can't quite see what is happening yet let's try another number.

```
X   = 1024 = 0000,0100,0000,0000b
&
X-1 =1023 = 0000,0011,1111,1111b
            0000,0000,0000,0000b
```

I don't see it yet let us try another number

```
X   = 8764 = 0010,0010,0011,1100b
&
X-1 =8763 = 0010,0010,0011,1011b
            0010,0010,0011,1000b
```

I see it! The bits starting on the right of the number are set to zero by the operation until the first 1 is met, after that all bits remain unchanged. The bit corresponding to the first 1 found is set to zero.

If there is only one bit then it will be set to zero by the operation. Powers of two only contain one set bit.

Try 5: Lets implement the algorithm in assembly language to see if we can get it to go faster. In reality the C++ compiler does a really good job so the improvement is unlikely to be great if at all.

```
bool power_of_two(unsigned int x)
{
      __asm
    {
        mov EAX,x    // AX=x  EAX is 32 bit version of 16 bit AX
        mov EBX,EAX  // BX=x
        dec EAX      // AX=x-1
        and EAX,EBX  // AX=AX & BX = x&(x-1)
        mov x,EAX    // x=AX
       }

       if(x==0) return(true); else return(false);
}
```

Try 6: Intel have created a single instruction popcount instruction. This is very fast and not available in standard C++ code. We can access it using inline assembly language as follows.

```
bool power_of_two(unsigned int x)
{
    __asm
    {
        mov EAX,x
        popcnt EAX, EAX
        mov x,EAX
       }
       if(x==1) return(true); else return(false);
}
```

We will stop here, however the next speed up might be to see if multiple popcount instructions can be run in parallel across multiple cores on the CPU processor. The ultimate performance (if you needed to do a lot of popcounts) might be to consider using a massively multi-core processor found in a GPU (Graphics Processor Unit).

Observation: There are many ways of solving the same problem, some are easy to formally prove as correct, others are easy to understand, other are really fast and others use very little working memory. There is no correct answer to the question but as a programmer all these additional constraints and options should inform your code design.

Aside: The above technique forms part of the method developed by Brian Kernighan's (of C fame) as a way to do a popcount.

```
unsigned int x=1023; // count the number of bits set in x
unsigned int c;      // c accumulates the total bits set in x
for (c = 0; x; c++)
{
  x &= x - 1;        // clear the least significant bit set
}
```

Addendum: I would be interested to know of other methods for detecting powers of two, so keep me posted if you find or invent one. One suggestion was to use a lookup table; this would require a big array even if we limited ourselves to 16 bits, but would be straightforward to implement and to verify.

Part 2: Using AND, XOR and OR

Enter the code shown below and run it, it should display the binary form of any number that you enter.

```
#include "stdafx.h"
#include <math.h>

void print_binary(char*,unsigned int);

int _tmain(int argc, _TCHAR* argv[])
{
	unsigned int x;

	do{
		printf("Enter a number:");
		scanf(" %u",&x);
		print_binary("Input :",x);

		// Add your code here

		print_binary("Output:",x);
	}while(1);

	return 0;
}

void print_binary(char text[],unsigned int x)
{
 printf("%s",text);
 for(int i=31;i>=0;i--)
 {
	 if(( (x>>i)&1)==0) printf("0"); else printf("1");
 }
 printf("\n"); // Newline
}
```

Consider the following truth tables for AND, OR and XOR

A	B	A.B AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A+B OR
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A⊕B XOR
0	0	0
0	1	1
1	0	1
1	1	0

Masking: When you AND two numbers you can use the set bits (1) to signify the bits that you wish to leave unchanged and the reset bits (0) that force the bit to zero.

If you have a 16 bit number and only want to look at the 8 least significant bits then you could AND the number with 255 (0000,0000,1111,1111b). You could try running the following code to see this idea in operation.

```
printf("Enter a number:");
scanf(" %u",&x);
print_binary("Input :",x);

x=x&255;

print_binary("Output:",x);
```

to invert all bits in a number you could XOR the number with another number that has all the bits set. So for 8 bits the code would look as follows.

```
printf("Enter a number:");
scanf(" %u",&x);
print_binary("Input :",x);

x=x^255;

print_binary("Output:",x);
```

OR can be used to force certain bits of choice to a set state. This code forces the last 8 bits to a set state.

```
printf("Enter a number:");
scanf(" %u",&x);
print_binary("Input :",x);

x=x|255;

print_binary("Output:",x);
```

Part 3: Some other twiddles

Using XOR to swap two values: To swap two variable values you normally need a third variable, something like the following...

```
int _tmain(int argc, _TCHAR* argv[])
{
	unsigned int x,y,temp;

	x=10;
	y=20;

	temp=y;
	y=x;
	x=temp;

	printf("X=%d, Y=%d",x,y);
}
```

In fact this is not the case, try the following code which makes use of the XOR operator ^. The code does not use a third variable to do the swap.

```
int _tmain(int argc, _TCHAR* argv[])
{
	unsigned int x,y;

	x=10;
	y=20;

	x^=y;
	y^=x;
	x^=y;

	printf("X=%d, Y=%d",x,y);
}
```

You could also use a stack e.g. PUSH X, PUSH Y, POP X and POP Y.

Calculate a location in screen memory: Have you ever wondered why a screen resolution is 1024x768 or 1280x960. Part of the answer lies in the fact that it is easier to multiply these numbers than say more obvious choices such as 1000x800. Screen memory starts at a base address and then each pixel is located side by side and then line by line. To access the screen memory on line 100, column 200 of a 1280x960 screen would require the following calculation

pixel_address=base_address+(100*1280)+200

A standard multiplication requires 16 (or 32) shift, bit check and then add options. However looking at the binary of 1280 it can be seen there are only 2 bits set (bits 8 and 10).

1280d= 10100000000b

So a much quicker way of doing the multiplication is to use to shift operations. Making the following observation

1280=1024+256=2^10 + 2^8

gives the following code that can multiply any number by 1280 using two shift operations and one add

```
int _tmain(int argc, _TCHAR* argv[])

{
	unsigned int x=100;

	x=(x<<10)+(x<<8);  // Multiply by (1024+256)=1280

	printf("X=%d",x);
}
```

Part 3: A few programming challenges.

Challenge 1: Add code to the program at the start of part 2 that inverts all the bits.

Challenge 2: Write code to invert only every second bit.

Challenge 3: Write code in the highlighted section that reverses the bits of the number you entered.

Submit your code (just the cpp file) for the highest number challenge you completed.

Final Thought... When you are writing code in high level languages, the experience of using low level code such as assembly languages gives you the knowledge to make informed choices about the data structures and algorithms you choose to use to solve the problem.