12-bit I/O

Design Pattern and I/O
Endianness

- When two parties wish to exchange information, then they need to agree on an ordering convention, if data being exchanged is too large to be sent in one piece.

- In computing, endianness refers to the byte or bit ordering of data stored in the computer memory or send over the network.

- We distinguish two orderings:
  - Big-endian order
  - Little-endian order
Big-Endian

- The most significant byte or bit (MSB) is stored at the memory location with the lowest address:

```
<table>
<thead>
<tr>
<th>4:</th>
<th>3:</th>
<th>2:</th>
<th>1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x45</td>
<td>0xA9</td>
<td>0xDD</td>
<td>0x78</td>
</tr>
</tbody>
</table>
```

```
α: 0x45
α+1: 0xA9
α+2: 0xDD
α+3: 0x78
```

Big-Endian
Little-Endian

- The least significant byte or bit (LSB) is stored at the memory location with the lowest address:

```
4: 3: 2: 1:
0x45 0xA9 0xDD 0x78
```

```
:α
:α+1
:α+2
:α+3
```

Little-Endian
What is the best for 12-bit I/O?

- Consider four 12-bit numbers: 1256, 382, 45, and 3409:

```
0xD51 0x2D 0x17E 0x4E8
```

- What is the best option to store these byte values?
Little-endian bit ordering:
Processing 4E8

0x4E8: 0 1 0 0 1 1 1 0 1 0 0 0

12-bit stream
0x172
Write Bits

write12Bits( CODE ):  

for i := 1 to 12  

do

if (CODE & 0x1)

then send 1 to output;

else send 0 to output;

CODE := CODE / 2;

don;
Read Bits

read12Bits():

read 12 bits from input to DATA[2];
Result := 0;
ByteIndex := 0; BitIndex := 8;
for i := 0 to 12

do

if ( DATA[ByteIndex] & (1 << (BitIndex - 1)) )

then Result := (1 << i) + Result;
BitIndex := BitIndex - 1;
if (BitIndex == 0 )

then ByteIndex := 1; BitIndex := 8;

od;
• **Intent:**

  • Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

• **Collaborations:**

  • Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.
Structure of an Object Adapter

Client → Target

Request()

Adapter

Request()

→ Adaptee

SpecificRequest()
The Class OBitStream

<table>
<thead>
<tr>
<th>OStream12Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>- fOStream : std::ofstream</td>
</tr>
<tr>
<td>- fBuffer : byte[32]</td>
</tr>
<tr>
<td>- fByteIndex : int = 0</td>
</tr>
<tr>
<td>- fBitIndex : int = 8</td>
</tr>
<tr>
<td>- init() : void</td>
</tr>
<tr>
<td>- finishWriteBits() : void</td>
</tr>
<tr>
<td>- writeBit0() : void</td>
</tr>
<tr>
<td>- writeBit1() : void</td>
</tr>
<tr>
<td>+ open( aFileName : const char * ) : void</td>
</tr>
<tr>
<td>+ close() : void</td>
</tr>
<tr>
<td>+ fail() : bool</td>
</tr>
<tr>
<td>+ flush() : void</td>
</tr>
<tr>
<td>+ operator&lt;&lt; ( aCode : int ) : OStream12Bits&amp;</td>
</tr>
</tbody>
</table>
The Object Adapter OStream12Bits

class OStream12Bits
{
private:
    std::ofstream fOStream;
    byte fBuffer[32];
    int fByteIndex;
    int fBitIndex;

    void init();    // initialize data members
    void finishWriteBit(); // complete write
    void writeBit0();   // write 0
    void writeBit1();   // write 1

public:
    OStream12Bits();
    OStream12Bits( const char* aFileName );

    void open( const char* aFileName );
    void close();
    bool fail();
    void flush();
    OStream12Bits& operator<<( int aCode );
};

OStream12Bits is an object adapter for an ofstream object.
The Constructors

OStream12Bits::OStream12Bits()
{
    init();
}

OStream12Bits::OStream12Bits( const char* aFileName )
{
    init();
    open( aFileName );
}
The init Method

```cpp
void OStream12Bits::init()
{
    for ( int i = 0; i < 32; i++ )
        fBuffer[i] = 0;

    fByteIndex = 0;
    fBitIndex = 8;
}
```
void OStream12Bits::writeBit0()
{
    fBitIndex--;  // Decrement the bit index
    finishWriteBit();  // Finish writing the bit
}

void OStream12Bits::writeBit1()
{
    fBuffer[fByteIndex] += 1 << (fBitIndex - 1);  // Shift 1 left by (fBitIndex - 1) bits and add to fBuffer
    fBitIndex--;  // Decrement the bit index
    finishWriteBit();  // Finish writing the bit
}
void OStream12Bits::finishWriteBit()
{
    if ( fBitIndex == 0 )
    {
        if ( fByteIndex == 31 )
        {
            fByteIndex++;
            flush();
        }
    }
    else
    {
        fByteIndex++;
        fBitIndex = 8;
    }
}
void OStream12Bits::open( const char* aFileName )
{
    fOStream.open( aFileName, ofstream::binary );
}

bool OStream12Bits::fail()
{
    return fOStream.fail();
}

void OStream12Bits::close()
{
    flush();
    fOStream.close();
}
void OStream12Bits::flush()
{
    fOStream.write( (char*)fBuffer, fByteIndex + (fBitIndex % 8 ? 1 : 0) );
    init();
}
The Output Operator

OStream12Bits& OStream12Bits::operator <<( int aCode )
{
    aCode = aCode & 0x0fff; // mask 12 lower bits

    for ( int i = 0; i < 12; i++ ) // write 12 Bits
    {
        if ( aCode & 0x01 ) // The current lowest bit is set.
            writeBit1();
        else
            writeBit0();
        aCode >>= 1; // Code := Code / 2
    }

    return *this;
}