CHAPTER 10
VIRTUAL FUNCTIONS - ALGEBRAIC CODING THEORY

10.1 Introduction

We will present the virtual function paradigm of C++ in an implementation of algebraic coding theory. Algebraic coding theory, also known as the theory of error correcting codes, was originated in 1950 by Richard Hamming, and has been an area of intense research and development for forty years. Error correcting codes are used throughout the computer and communications industries; modern communication and computing could not exist without them. We will use them to encrypt text files and to improve the pattern recognition capabilities of a neural net. The neural net application, originated by the author, will be developed in Chapter 11. Modulo two vectors and matrices are the underlying mathematical structures for algebraic coding theory and the bit vectors and matrices of Chapter 9 are the classes needed to implement the theory in C++. I first implemented them using the **modulo_number** class of Chapter 3 and a vector-matrix implementation for modulo two numbers that closely parallels the implementation in Chapter 5. Using an entire byte to store a zero or one and ordinary modulo arithmetic together yielded slow code; the performance was greatly improved by bit-mapping the zeros and ones and using the bitwise operations that are part of **C**.

10.2 The Underlying Principle of Error Correcting Codes

For the introductory discussion, we will focus on the original Hamming 4-7 codes. A *nibble* is the computer jargon name for a 4 bit pattern or half a byte. There are sixteen different nibbles. If we select a particular nibble and change one bit of that nibble, then the resulting 4 bit pattern is also a nibble. If we were transmitting nibbles over a communication line and all sixteen nibbles were *acceptable* to send and receive, a change of one bit would go undetected. We want a bit change on an acceptable pattern to yield an unacceptable pattern. If we want to transmit any of the possible sixteen 4 bit patterns, we must encrypt the 4 bit pattern into a bit vector with more than 4 components in such a manner that makes it possible to determine a one bit error has been made in a transmission. The Hamming 4-7 coding technique encrypts the entire set of sixteen nibbles into a subset, of size sixteen, of the 7-bit vectors. These special 16 7-bit vectors, called the codewords, are chosen in such a manner that the *Hamming distance* between any two of them is at least three.

What is Hamming distance? The Hamming distance between any two bit vectors of the same size is the number of locations where they differ. Hamming distance satisfies the triangle inequality: **HD(u,w) <= HD(u,v) + HD(v,w)** in which **u**,**v**,**w** are bit vectors of the same size. If **u** and **w** were distinct codewords of length 7 and **v** were a 7-bit vector at a distance of 1 bit from **u**, then this inequality produces **HD(v,w) >= 2**. Because of this, if a bit is changed in a codeword **u** to yield a 7 bit vector **v**, then **v** will be at least two bits away from all other codewords. This permits us to associate the 7-bit vector **v** uniquely to the codeword **u**.

The secret to success in designing error correcting codes is to choose a particular size **n** for a set of bit vectors and a positive integer constant **MD**, and then to select a special subset of the **n**-bit vectors in order that any two vectors in the subset are at least **MD** (minimum distance) away from each other; this subset is called the codewords. Finding all of the above is by no

means trivial, and has been the subject of many years of research. Associating any **n**-sized bit vector to a unique codeword is called *error correcting*. Because of the triangle inequality, this constant **MD** determines whether we can associate a **n**-bit vector with a unique codeword. Depending on whether **MD** is odd or even, we have the following:

(1) **MD** is odd: we can correct a bit vector within **MD/2** from a codeword
(2) **MD** is even: we can correct a bit vector within **(MD/2)-1** from a codeword

For the Hamming 4-7 codewords (7 bit vectors), we can uniquely correct 7-bit vectors that are a distance 1 away from a codeword. For another set of codewords, called maximal length shift register codes of dimension 127, we can correct a 127-bit vector that is a distance of 31 (64/2 - 1) or less from a codeword. This is significant and will be discussed in a later section of this chapter.

We have to be sufficiently close to a codeword in order to associate a bit vector uniquely to that codeword; sometimes this is not possible. In the case of maximal length shift register codes, *sufficiently* close is quite liberal. Also, we could start with a code word and change enough of its bits, putting it within the *correcting distance* of another codeword. For example, if we change two bits of a Hamming 4-7 codeword, then we are 1 bit away from another Hamming 4-7 codeword and 2 bits away from the original. We would end up correcting to the *wrong* codeword. We have to live with this in error correcting codes. Also, a codeword could have enough bits changed to yield another codeword; in that case, we would not even realize that bit changes were made. For Hamming 4-7 codes, we can recognize that a codeword has been changed if either 1 or 2 bits have been changed; we cannot tell whether it was 1 or 2 though. If we have a non-codeword 7-bit vector, we can correct it uniquely to a codeword. This correction will be correct if only one bit actually changed. Otherwise, we are correcting it to the wrong codeword. These again are the caveats of error correcting codes.

10.3 Hamming 4-7 Codes

10.3.1 Parity Matrix Equation and Equivalence Classes

In order to understand some of the basics of coding theory, we will completely develop a set of Hamming 4-7 codes. We must take the sixteen possible nibbles (4-bit vectors) and map them, 1-1, into a subset of the 7-bit vectors. Each nibble will be mapped to a unique 7-bit vector. This special subset, consisting of sixteen vectors, is called the subset of codewords. We can write the encryption formula for this mapping as:

$$\mathbf{H}(\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}, \mathbf{n_4}) = (\mathbf{p_1}, \mathbf{p_2}, \mathbf{n_1}, \mathbf{p_3}, \mathbf{n_2}, \mathbf{n_3}, \mathbf{n_4}), \text{ where}$$

$$\mathbf{p_1} = \mathbf{n_1} + \mathbf{n_2} + \mathbf{n_4} \pmod 2$$
$$\mathbf{p_2} = \mathbf{n_1} + \mathbf{n_3} + \mathbf{n_4} \pmod 2$$
$$\mathbf{p_3} = \mathbf{n_2} + \mathbf{n_3} + \mathbf{n_4} \pmod 2$$

The above equations are in modulo 2 arithmetic, and the ordering of the **p**'s and **n**'s in the

formula defining **H** does play a crucial role in the mechanics of error correction. Adding the **p**'s to both sides of the above equations and using the identity **x + x** = **0** of modulo 2 arithmetic, we get the following equations:

$$\mathbf{p_1 + n_1 + n_2 + n_4 = 0} \pmod 2$$
$$\mathbf{p_2 + n_1 + n_3 + n_4 = 0} \pmod 2$$
$$\mathbf{p_3 + n_2 + n_3 + n_4 = 0} \pmod 2$$

If any 7-bit vector is labeled as **(p_1, p_2, n_1, p_3, n_2, n_3, n_4)**, and with this labelling satisfies the above system of equations, the 7-bit vector is a codeword because it could have been generated with the equivalent above formulas that define the **p**'s in terms of the **n**'s. This system of equations can be written as the following matrix equation:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ n_1 \\ p_3 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 10.1 - Parity Check Equation

The 3 by 7 matrix, denoted by **P**, in Figure 10.3 is called the *Hamming parity check matrix*. A 7-bit vector is a codeword in our set if and only if it satisfies this matrix equation.

The following discussion requires a working knowledge of basic linear algebra. Looking at **P**, we see that it has rank 3 (number of linearly independent rows in a matrix). The solutions to the above homogeneous matrix equation comprise a vector subspace of 7-bit vectors, called the null space of **P**, denoted by **Null(P)**. A major theorem of linear algebra states:

dim(Null(P)) + rank(P) = dim(Domain(P)).

The domain of **P** is the entire vector space of 7-bit vectors, yielding **Domain(P)** equal to 7. Thus, the null space of **P**, the vector space of codewords, has dimension 4. Since the coefficients are in the integers modulo 2, the total number of codewords is sixteen (2 to the power 4). This agrees with the original formula for generating codewords from nibbles, since there are sixteen distinct nibbles and distinct nibbles get mapped to distinct codewords. It is aesthetically pleasing and mathematically useful that the Hamming 4-7 codewords are exactly the null space

of a matrix.

The distributive law for matrix multiplication, **P*(C + D) = P*C + P*D**, shows that the sum of two codewords will also be a codeword. Also, **C + C = 0**, the zero vector, because of the property of modulo 2 arithmetic (**1 +1 = 0**); any codeword (actually any modulo 2 vector) is the additive inverse of itself. This shows that the set of Hamming 4-7 codewords is a mathematical group properly contained within the full group of 7-bit vectors. This group contains 16 vectors called the codewords.

A proper subgroup of a group can be used to partition the entire group up into a disjoint union of subsets, called *equivalence classes*. These equivalence classes can be produced by taking an element not in the subgroup (of codewords), and adding this element to each element in the subgroup. With the correct selection of non-codewords, we can produce distinct equivalence classes. For our group of codewords, we will use the *epsilon-i* vectors, denoted by $\mathbf{E_i}$, to produce these equivalence classes. $\mathbf{E_i}$ has a **1** in the **i**th component and zeros everywhere else. Denote by $\mathbf{C_i}$ the set of vectors obtained by adding $\mathbf{E_i}$ to each codeword. No $\mathbf{E_i}$ is a codeword because **P,** $\mathbf{E_i}$ is the **i**th column of **P**, a nonzero vector. From group theory it follows that no element of $\mathbf{C_i}$ is a codeword, all elements of $\mathbf{C_i}$ are distinct, and thus there are sixteen different vectors in $\mathbf{C_i}$. The big question now is: Are $\mathbf{C_i}$ and $\mathbf{C_j}$ distinct for **i** and **j** distinct? Note that $\mathbf{E_i + E_i = 0}$ and thus when we add two vectors in $\mathbf{C_i}$, we are in total adding two codewords and two copies of$\mathbf{E_i}$. Thus the sum of two vectors in $\mathbf{C_i}$ is a codeword. When we take the sum of a vector in $\mathbf{C_i}$ with a vector in $\mathbf{C_j}$, the result is a codeword plus the sum of $\mathbf{E_i}$ and $\mathbf{E_j}$. Can this sum of a vector in $\mathbf{C_i}$ with a vector in $\mathbf{C_j}$ be a codeword? When we multiply the sum $\mathbf{E_i + E_j}$ by **P**, the result is the sum of the **i**th and **j**th columns of **P**. Since the columns of **P** are distinct, the only way we can have a product being the zero vector is if **i** and **j** are the same. This shows the classes $\mathbf{C_i}$ are distinct for different $\mathbf{E_i}$.

It follows that we have partitioned the 128 7-bit vectors into 8 disjoint sets, and each 7-bit vector is either a codeword or it is in some $\mathbf{C_i}$, exactly one bit away from a unique codeword. $\mathbf{C_i}$ is the 7-bit vectors obtained by adding an *error* of one bit in the **i**th component of the set of codewords. If a 7-bit vector is not a codeword, we associate it with the unique codeword that it is one bit away from.

10.3.2 Correcting a 7-bit Vector

Let $\mathbf{P_i}$ denote the **i**th column of **P**. Upon inspection you can verify the bit pattern of $\mathbf{P_i}$ has the integer value **i**. **P** times any 7-bit vector must yield the zero vector or a column of **P** since the columns of **P** totally exhaust all of the seven 3-bit nonzero vectors. This is the intricacy in designing the matrix approach to Hamming codes. Assume that we have a seven bit vector **Y** that yields:

$$\mathbf{P * Y = P_i} \quad \textbf{for some i.}$$

Determining the **i** and adding $\mathbf{E_i}$ to **Y**, we get:

$$\mathbf{P * (Y + E_i) = P * Y + P * E_i = P_i + P_i = 0}$$

Thus $\mathbf{Y + E_i}$ is the codeword 1-bit away from **Y**. Note that the bit representation of $\mathbf{P_i}$ has value **i**, and this instructs us to change the **i**th bit of **Y** to get the nearest codeword to **Y**. Adding $\mathbf{E_i}$ to **Y** is an algebraic vector way to accomplish this. This parity matrix **P** multiplication also shows that if **X** is a codeword, then $\mathbf{X + E_i + E_j}$ is a codeword only if **i = j**. This is because the **i**th and **j**th columns of **P** only sum to zero when **i = j**. Thus a two bit change to a codeword does not yield another codeword, showing that two distinct codewords must be at least three bits apart in Hamming distance.

10.3.3 Hamming 4-7 Generating Matrix

Since we have a bit vector and bit matrix implementation, we naturally want to encrypt nibbles into 7-bit vectors using the functionality of this implementation. The following figure defines the Hamming generating matrix for this set of codes:

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix}$$

Figure 10.2 Generating Matrix Equation for Hamming 4-7 Codes

The Hamming 4-7 codes are not implemented in this book, but the Hamming 7-11 codes are. It will be left as an exercise to implement and apply the Hamming 4-7 codes to a cryptography problem. The Hamming 4-7 C++ implementation closely parallels the Hamming 7-11 implementation.

10.4 Hamming 7-11 Codes

The codes implemented in this chapter encrypt 7-bit vectors into a set of vectors of size 11, 15, or 127. 7-bit codes were chosen because the ASCII character set has 128 elements in it and there exists easily described codes for encrypting 7-bit vectors. These 7-bit encrypting codes will be used in Chapter 11 to encrypt the 7-bit patterns of ASCII characters into sets of codewords of a higher size. The mathematical theory for Hamming 7-11 codes is not as clean as that for the 4-7 codes, but the C++ implementation is of the same difficulty.

10.4.1 Generating and Parity Matrices for the Hamming 7-11 Codes

We will need a 4 by 7 matrix to generate the parity bits for the Hamming 7-11 codes and a 4 by 11 matrix for codeword recognition and error detection. The set of codewords will have cardinality 128 (2^7), with the space of 11-bit vectors partitioned into 16 (2^4) equivalence classes. Eleven of these equivalence classes are generated by adding the $\mathbf{E_i}$'s to the subgroup of codewords; these eleven equivalence classes comprise the 11-bit vectors that are exactly one bit away from a codeword. The codeword set brings the count of equivalence classes to twelve, leaving four classes. Because of the equivalence class theory, the four remaining equivalence classes comprise vectors that are at least two bits away from a codeword. Each 11-bit vector in these 4 classes is two bits away from more than one codeword, making impossible a unique correction. Multiplication of an 11-bit vector by the parity check matrix will yield a column of the parity check matrix, a zero vector, or a vector whose bit pattern has value from 12 through 15. The **i**th column of **P** indicates the vector can be changed to a codeword by changing the **i**th bit, the zero vector indicates a codeword, and a bit pattern with value larger than eleven indicates the vector is two bits from the closest codeword. The parity matrix and equivalence class theory show the minimum distance between two codewords is three, permitting positive identification of one bit errors. If a codeword has two bits changed, the parity check equation will determine the resulting vector is not a codeword. Depending on what the parity check matrix multiplication yields, it will be recognized correctly as a two bit change or incorrectly as a one bit change. If a one bit change is the conclusion, the vector will be corrected to the codeword one bit away, which does differ from the original. To make it crystal clear, some two bit errors are detected as two bit errors while others are detected as 1 bit errors. For the Hamming 4-7 codes, because every non-codeword bit vector is one bit away from a codeword, all two bit errors are interpreted as one bit errors. The bottom line is Hamming 7-11 codes for 11-bit vectors are equivalent in performance to the 4-7 codes for 7-bit vectors; if a one bit error occurs, it will be correctly diagnosed, otherwise you take your chances.

.

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \end{bmatrix}$$

Figure 10.3 Generating Matrix for Hamming 7-11 Codes

In Figure 10.3, we have the matrix that generates the four parity check bits for the 11-bit

codewords. These four bits are combined with the original 7-bit vector to yield the 11-bit codeword: ($\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{n_1}$, $\mathbf{p_3}$, $\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_4}$, $\mathbf{p_4}$, $\mathbf{n_5}$, $\mathbf{n_6}$, $\mathbf{n_7}$). We could have defined the generating matrix to be 11 by 7, including the n's in the product vector. This 4 by 7 matrix is used in the C++ implementation of Hamming 7-11 codes.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ n_1 \\ p_3 \\ n_2 \\ n_3 \\ n_4 \\ p_4 \\ n_5 \\ n_6 \\ n_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 10.4 Parity Check Equation for Hamming 7-11 Codes

If an 11-bit vector satisfies the above equation, it is a codeword. If the parity matrix when multiplied against an eleven bit vector yields the **i**th column of **P**, we interpret the 11-bit vector to be a codeword that has had its ith bit changed. We get the codeword one bit away by changing the **i**th bit of the vector. If the parity matrix multiplied against an 11-bit vector yields one of the four vectors not making up the columns of **P**, we say that we cannot correct the vector because it is two bits away from more than one codeword and not one bit away from any codeword.

Two references that have excellent discussions on these kinds of cyclic codes are Gamble[] and Solomon (author) - Balakrishnan (editor) [].

10.5.1 The Shift Register

The key ingredient for generating a group (mathematical meaning) of shift register codes is a difference equation. In the C++ source code, we will build two groups, each with cardinality 128, of shift register codes: one group of length 15 with minimum distance of 5 among codewords and another of length 127 with minimum distance of 64 between codewords.

Because of its simplicity, we will study the shift register, with three binary storage elements and difference equation $\mathbf{a_n + a_{n+1} = a_{n+3}}$, shown below:

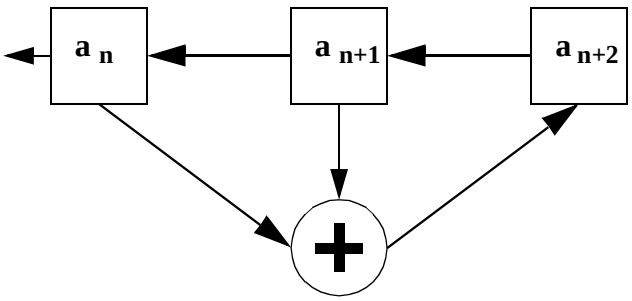

Figure 10.5 Shift Register for $\mathbf{a_n + a_{n+1} = a_{n+3}}$

The coefficients, called the feedback coefficients, for the difference equation of this shift register are **1**, **1**, **0**, corresponding respectively to $\mathbf{a_n}$, $\mathbf{a_{n+1}}$, $\mathbf{a_{n+2}}$.

For a choice of initial conditions, $\mathbf{a_0}$, $\mathbf{a_1}$, $\mathbf{a_2}$, this difference equation generates an infinite sequence of **0**'s and **1**'s. We will generate eight finite sequences of seven bits from the eight possible 3-bit vectors, which represent initial condition sets, and choose these as our group of codewords. This group is:

(1) 1 1 1 0 0 1 0
(2) 1 1 0 0 1 0 1
(3) 1 0 0 1 0 1 1
(4) 0 0 1 0 1 1 1
(5) 0 1 0 1 1 1 0
(6) 1 0 1 1 1 0 0
(7) 0 1 1 1 0 0 1
(8) 0 0 0 0 0 0 0

The difference equation actually represents the following system of four equations in seven unknowns when we use it to generate 7-length sequences:

$$
\begin{aligned}
a_0 + a_1 + a_3 &= 0 \\
a_1 + a_2 + a_4 &= 0 \\
a_2 + a_3 + a_5 &= 0 \\
a_3 + a_4 + a_6 &= 0
\end{aligned}
$$

Remember the solutions of a homogeneous system constitute a vector space, thus a group under addition. Because these 7-bit vectors are determined by the first three bits, we have eight possible initial condition sets, yielding eight solutions of the difference equations. Linear algebra theory for a system of seven unknowns with matrix rank four yields a null space of dimension three, with a total of eight codewords. We will refer to this group as a group of shift register codes associated with the above difference equation. Because the *Hamming weight* (number of nonzero bits) of each nonzero codeword is four, the sum of any two codewords is another codeword, the distance between any two codewords is Hamming weight of their sum, we have a distance of four always between any two codewords. If a 7-bit vector is one bit away from a codeword, that codeword is the unique closest codeword to the vector, and if a 7-bit vector is two bits away from a codeword, that vector is not one bit away any codeword. In the latter case, a codeword is not associated with this vector since it is outside of the correcting radius, (one in this case) of any codeword.

The electronic shift register of Figure 10.5 has a state consisting of a 3-bit vector stored in the three boxed bit memory locations and a feedback vector used to add, modulo two, a fixed subset of the stored bits. The arrows leading from the square boxes to the circled adder indicate that the contents of these two bit locations are to be added, and then the contents of the three boxes are shifted to the left with the result of the adder being placed in the storage area labeled $\mathbf{a_{n+2}}$. The content of $\mathbf{a_n}$ is possibly stored in some other container. We will store it in the codeword we are generating. Again this is a *machine* representation, which is an electronic component, of our difference equation. We must initially place $\mathbf{a_0}$, $\mathbf{a_1}$, $\mathbf{a_2}$ in the memory locations of the shift register, calculate $\mathbf{a_3}$, and put it in the rightmost box, shifting off $\mathbf{a_0}$ from the leftmost box. Then we calculate $\mathbf{a_4}$, put it in the rightmost box, and shift off $\mathbf{a_1}$ simultaneously. This is the mechanism used to encrypt a 3-bit vector into a 7-bit vector. This group of eight 7-bit vectors makes a viable set of codewords that can correct one bit errors.

10.5.2 Maximal Length Shift Register Sequences

If we choose an **n**-stage shift register (or its corresponding nth order difference equation), we can encrypt the entire set of **n**-bit vectors into a vector space of larger size, depending upon the length of the finite sequences generated. As long as this length is larger than or equal to **n**, we will have a group of distinct codewords (because of the **n**th order difference equation). Generating sequences of length **n** does nothing for us; we want to generate sequences of a length sufficiently large that we have a *desirable* minimum weight, **MD**, among the codewords. Because of the group property, **MD** will be the minimum distance between codewords. No fancy mathematics is needed to find this distance; simply run through the codewords calculating the minimum weight. Since the distance between any two codewords (generated by the shift register) is the Hamming weight of their sum and

their sum is also a codeword, we check the weights of the 2^n codewords. This is preferable to calculating the $(2^n)^2 /2 - 2^n$ distances for all possible pairs of codewords. For the above 3-stage shift register example, the minimum distance is four, the correcting distance one. Advanced mathematical theory can eliminate a great deal of the guess work in selecting a set of shift register codes and calculating the correcting distance. Of particular interest is the set of codes grouped together under the title of maximal shift register sequences. Every shift register (given a non-zero initial state) will generate sequences that are periodic, and the maximum period is $2^n - 1$ where n is the size of the shift register. It is not always possible to achieve this length, but it can be done when $2^n - 1$ is a prime number. The low order cases are for $n = 3$, $n = 5$, and $n = 7$ with respective periods 7, 31, and 127. To achieve this, we must select the underlying difference equation with an irreducible modulo two *characteristic polynomial*. The characteristic polynomial of $a_k + c_1 a_{k+1} + c_2 a_{k+2} + ... + c_m a_{k+m} = a_{m+1}$ is:

$$p(x) = 1 + c_1 x^1 + c_2 x^2 + ... + c_m x^m + x^{m+1}$$

The size of the shift register for this difference equation is $m+1$. Note that the coefficient of a_k is always 1. In the three stage example presented above, the irreducible polynomial was $p(x) = 1 + x^1 + x^3$. For the case $n = 7$ presented in the source code for this chapter, the feedback coefficients are **1**, **1**, **1**, **1**, **1**, **1**, **0**. The difference equation, with characteristic polynomial irreducible modulo two, associated with this shift register is:

$$a_k + a_{k+1} + a_{k+2} + a_{k+3} + a_{k+4} + a_{k+5} = a_{k+7}$$

By using one entire nonzero sequence of length 127, we can select the codeword generated by anyone of the possible 127 nonzero 7-bit vectors. Each possible 7-bit pattern occurs somewhere in the 127 bit pattern, including cyclic wrap around. We will chose all 127 nonzero sequences of length 127 and add the zero sequence to produce a group of codewords. These codewords are shifts of one another in the proper order, constituting a group because they satisfy the same difference equation (and associated 120 by 127 matrix homogeneous equation). The significant property of maximal shift register sequences is that their Hamming weight is $(2^n - 1)/2 + 1$. For $n = 7$, the Hamming weight of each codeword is 64. Because of this, a 127-bit vector within 31 bits of a codeword can be associated (corrected) uniquely to that codeword. This is significant. There is overhead when encrypting a 7-bit vector into a 127-bit codeword, but the error correction capability is a whopping 25%.

10.5.3 Cyclic Codes of Non-maximal Length

A seven stage shift register was also chosen with more modest error correction capability. The feedback coefficients for this one are **1**, **1**, **0**,**1**, **0**, **0**, **0**. Because the polynomial associated with these coefficients is not irreducible, these will not be maximal length sequences (with period 127). The mathematical theory for maximal length sequences does not apply. Nevertheless, we generate all 127 nonzero sequences of length fifteen, add the zero sequence, obtaining a group of 15-bit codewords, with five the smallest Hamming weight among these codewords.

Thus there is a distance of at least five between any two codewords, permitting two bits of error correction. This is a significant improvement over the Hamming 7-11 codes at a cost of only four more encryption bits. This set of 15-bit codewords can correct at a 13% bit error rate, while the 127-bit codewords can correct at 25%, albeit with many more bits.

10.6 Implementation of Coding Theory in C++

The Hamming codes and the cyclic shift register codes are two distinct kinds of error correcting codes, especially at the implementation level. The three sets of codes implemented in this chapter all encrypt 7-bit vectors. The Hamming code, implemented here, encrypts 7-bit vectors into 11-bit codewords while the two sets of shift register codes implemented encrypt 7-bit vectors into 15 and 127 bit codewords respectively. Since the function of different error correcting codes is basically the same, one expects functionality to be the same across all coding theory varieties. The inner workings of the error correction should be transparent to the application. A method to achieve application transparency is to use runtime polymorphism to determine the type of error correcting codes being used and to invoke automatically the correct functionality. In Chapter 11, error correcting codes are used by a neural net to recognize patterns. If an object representing an entire set of codewords is passed to the neural net and the functionality associated with the coding theory classes was implemented properly, the neural net does not have to know the exact kind of error correcting being performed.

For reasons to be discussed below, a base class, named **Base_Code**, is created, but no objects of its type will be instantiated. Then two classes, named **h7_11** and **cyclic7**, are derived from this class. Even though objects will never be created of type **Base_Code**, we do use variables of type **Base_Code*** to store pointers. Storing derived class pointers in base class pointer variables is fundamental to runtime polymorphism.

10.6.1 Header File for Error Correcting Codes

```
#ifndef CODING_H
#define CODING_H
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include "tracer.h"
#include "m2_d7.h"
#include "sreg7.h"
class Base_Code {
public:
    //  virtual Base_Code() {};
    virtual int Code_Length() const  = 0;
    virtual int Correcting_Distance() const  = 0;
    virtual bit_vector closest_codeword(const bit_vector&) const  = 0;
    virtual bit_vector generate(const bit_vector&) const  = 0;
    virtual ascii_vector decode(const bit_vector&) const  = 0;
```

```
};

class h7_11 : public Base_Code {
public:
    static bit_matrix p_matrix;
    static bit_matrix gen_matrix;
    static int code_length;
    static int log_group_size;
    int Code_Length() const  { return code_length; }
    int Correcting_Distance() const  { return 1; }
    bit_vector closest_codeword(const bit_vector& mdv) const;
    ascii_vector decode(const bit_vector& mdv) const;
    int syndrome(const bit_vector& mdv) const;
    bit_vector generate(const bit_vector& mdv) const;
    h7_11() {  };
};

class cyclic7_15 : public Base_Code {
private:
    int word_length;
    int code_length;
    shf_reg_7 shreg;
    bit_vector *code_ptr_15;
public:
    cyclic7_15(const bit_vector&);
    cyclic7_15();
    int Code_Length() const  { return code_length; }
    int Correcting_Distance() const  { return 2; }
    bit_vector *contents_ptr() const  { return code_ptr_15; }
    int minimum_distance() const;
    int correct_code(const bit_vector&) const;
    bit_vector closest_codeword(const bit_vector& mdv) const;
    ascii_vector decode(const bit_vector& mdv) const;
    bit_vector cyclic7_15::generate(const bit_vector& mdv) const;
};

class cyclic7_127 : public Base_Code {
private:
    int word_length;
    int code_length;
    shf_reg_7 shreg;
    bit_vector *code_ptr_127;
public:
    cyclic7_127(const bit_vector&);
    cyclic7_127();
    int Correcting_Distance() const  { return 31; }
    int Code_Length() const  { return code_length; }
    bit_vector *contents_ptr() const  { return code_ptr_127; }
    int minimum_distance() const;
    int correct_code(const bit_vector&) const;
```

```
    bit_vector closest_codeword(const bit_vector& mdv) const;
    ascii_vector decode(const bit_vector& mdv) const;
    bit_vector cyclic7_127::generate(const bit_vector& mdv) const;
};

#endif CODING_H
```

10.6.2 Virtual Function Paradigm of C++

The self-identification of type is accomplished through the virtual function paradigm of C++, and the implementation of virtual functions is dependent upon pointers to objects invoking member functions, not objects represented by variables invoking member functions.

To implement runtime polymorphism, we must have an inheritance hierarchy that includes a base class and derived classes, and then declare member functions virtual that will be redefined in derived classes. In all our coding theory implementations, there is a function with prototype:

```
bit_vector closest_codeword(const bit_vector& mdv).
```

This function returns the closest codeword to the argument **mdv** with some caveats. When a **bit_vector cw** is returned by **closest_codeword**, it should be verified that **mdv** is within the valid correcting distance of **cw** for that set of codes. In the case of Hamming codes, if there is no codeword within distance one of a bit_vector **mdv**, **closest_codeword** returns the null codeword. The null codeword acts as the *exception handler* in this case; the applications program should use the **Hamming_Distance** function and the member function with prototype **int Correcting_Distance()** function to determine if the bit_vector **mdv** is within **Correcting_Distance** of the returned codeword. If not, we do not accept the codeword returned by **closest_codeword**, and make the decision that correction is not possible. For shift register codes, we do not use a null codeword as an exception handler, but we do check the *correcting distance* validity using functions with the same identifiers as the Hamming code function identifiers. The application program calls the functions **closest_codeword** and **Correcting_Distance** with a coding theory object. It does not need to know the actual error correcting code being used. A code bite from the next chapter that illustrates the above is:

```
int choice = 1; // choose 1 for illustrative  purposes
Base_Code*  bcptr;
bit_vector mod2_output;
// create a set of  codewords and store the pointer to them in a base class pointer
switch choice {
case 1:
    bcptr = new h7_11;  // construct the hamming7_11  codes, and assign pointer to
    // base   class  pointer variable
    mod2_output  =  ...  //  produce it somehow, should be an eleven bit
    // vector to correspond with h7_11
    break;
```

```
case 2:
    static int xp[7] = { 1, 1, 1, 1, 1, 1, 0 };
    bit_vector tempvec1(xp, 7); // bitmap xp
    bcptr = new cyclic7(tempvec1, 127, 31);
    mod2_output = ... // produce a 127-bit vector to correspond    to codewords
}
//  the following is independent of what  error correcting codes we are using
//  the error codes must be the same length as mod2_output though
//  view the above as input to below
int result;
bit_vector correction = bcptr->closest_codeword(mod2_output);
if (mod2_output.hamming_distance(correction) <= bcptr->Correcting_Distance() )
    result =  (char) (bcptr->decode(correction) ); //   decode decrypts a  codeword
    // back to the original 7 bit vector, then a char operator converts it to a
    // char variable
else result = -1;   // negative indicates no correction made
```

When the above code is compiled, it is not relevant what type of error-correcting pointer is assigned to **bcptr**. The functions **closest_codeword**, **Correcting_Distance**, and **decode** are declared **virtual** in **Base_Code**, with distinct code implementations defined in the **h7_11** and **cyclic7** classes. At runtime, the actual pointer type stored in bcptr will determine (indirectly, to be explained below) what versions of the virtual functions are called. The exact type of codes used are transparent to the bottom half of the above fragment; it will be able to determine a value for **result**, hopefully different from **-1**. This is the beauty of virtual functions.

To achieve this, the application program must call, with a pointer, member functions that are declared virtual. Furthermore, this pointer is usually stored in a base class pointer variable, even though the original address was a pointer to a derived class object. When a base class pointer invokes a member function, the original pointer type (possibly a derived class pointer type) is determined, and then the member function for the actual type in the derivation hierarchy is called. Study the above code fragment to see how this is done.

In the above, we create with new two sets of codewords, each of a different derived class. We keep a handle on one these two objects by assigning its pointer into a pointer variables of the base class type, **Base_Code***. IT IS LEGAL TO ASSIGN DERIVED CLASS POINTERS INTO BASE CLASS POINTERVARIABLES.

For illustrative purposes, let's discuss he function **Correcting_Distance**, a member function of both derived classes **h7_11** and **cyclic7**, that was declared virtual in **Base_Code**. The key is **Correcting_Distance** is declared in **Base_Code** as:

```
virtual int Correcting_Distance() const = 0;
```

The modifier **virtual** makes **Correcting_Distance** polymorphic. If the function **Correcting_Distance** is called by a pointer, then the actual object stored at the address referred to by the pointer determines which version of a virtual function is called. The type of pointer variable calling the virtual function does not determine which version of **Correcting_Distance** is called. In the above, if the address of a **cyclic7** code were stored in **bcptr**, **bcptr->Correcting_Distance()** would invoke the **Correcting_Distance** defined in the **cyclic7** class. Since the address of a **h7_11** code was stored in **bcptr**, **bcptr->Correcting_Distance()** invokes

the **Correcting_Distance** defined in the **h7_11** class. It does not matter that both **bcptr** is a **Base_Code** pointers. The beauty of this is that a function can be written that will be able to interchangeably use the functionality of any of the derived classes. All that is needed is to call these functions with base class pointers. The applications function is able to use objects of these different derived classes in a *plug compatible* way. This will be illustrated in the next chapter in which a back-propagation neural net uses these plug compatible error-correcting codes.

10.6.3 Virtual Function Table

If a class is declared to be virtual, then the storage for the data members no longer is the sum total of the storage for each individual data member. An extra unseen data member, called the pointer to the ***virtual function table***, is added to every object instantiated in the inheritance hierarchy for which their is a virtual function declared. At the highest point in the inheritance tree where some virtual function is declared, this inclusion will begin. One virtual function will trigger it, and only one virtual function table pointer data slot will be needed, no matter how many functions are declared virtual. For each class in the virtual hierarchy, there will be an array of functions pointers that point to the correct virtual member functions for that class. When an object is created, the address (pointer to an array of pointers) of the virtual function table is stored as the hidden data member for that object.

At compile time, if a virtual function is called by a pointer variable, no function is bound to that pointer variable. Rather, it is (through indirection) directed to invoke the proper function via the virtual function table. At runtime, objects of different classes in the hierarchy can have their addresses stored in this pointer, and at these addresses we will find different virtual function table addresses. This is the means by which runtime polymorphism is achieved.

When a variable (not a pointer) of a class invokes any function, the function is bound to that variable at compile time. Also, if a pointer invokes a function that is not virtual, then the function that corresponds to that pointer type is bound to the pointer at compile time. Complie time binding obviously makes for faster execution, but the power programming of runtime polymorphism is lost. Remember, blazing speed is not always necessary.

10.7 C++ Files

10.7.1 File *coding.h*

```
// file coding.h
#ifndef CODING_H
#define CODING_H
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include "tracer.h"
#include "m2_d7.h"
```

```
#include "sreg7.h"

class Base_Code {
public:
     virtual ~Base_Code() {};
   virtual int Code_Length() const = 0;
   virtual int Correcting_Distance() const = 0;
   virtual bit_vector closest_codeword(const bit_vector&) const = 0;
   virtual bit_vector generate(const bit_vector&) const = 0;
   virtual ascii_vector decode(const bit_vector&) const = 0;
};

class h7_11 : public Base_Code {
     int correcting_distance;
public:
     static bit_matrix p_matrix;
     static bit_matrix gen_matrix;
     static int code_length;
     static int log_group_size;
     h7_11() { correcting_distance = 1; }
     int Code_Length() const { return code_length; }
     int Correcting_Distance() const { return correcting_distance;}
     bit_vector closest_codeword(const bit_vector& mdv) const;
     ascii_vector decode(const bit_vector& mdv) const;
     int syndrome(const bit_vector& mdv) const;
     bit_vector generate(const bit_vector& mdv) const;
     ~h7_11() {};
};

class cyclic7 : public Base_Code {
private:
   int word_length;
   int code_length;
   shf_reg_7 shreg;
   int correcting_distance;
   bit_vector* code_ptr;
public:
   cyclic7(const bit_vector&, int, int);
   ~cyclic7();
   int Code_Length() const { return code_length; }
   int Correcting_Distance() const { return correcting_distance; }
   bit_vector* contents_ptr() const { return code_ptr; }
   int minimum_distance() const;
   int correct_code(const bit_vector&) const;
   bit_vector closest_codeword(const bit_vector& mdv) const;
   ascii_vector decode(const bit_vector& mdv) const;
   bit_vector cyclic7::generate(const bit_vector& mdv) const;
};

#endif CODING_H
```

10.7.1 File *coding.cpp*

```
// file coding.cpp
// implementation of hamming7_11 class and  some cyclic  codes

#include "coding.h"

int h7_11::code_length = 11;
int h7_11::log_group_size = 7;
static int phamming[4][11] = { {1,0,1,0,1,0,1,0,1,0,1},
    {0,1,1,0,0,1,1,0,0,1,1}, {0,0,0,1,1,1,1,0,0,0,0},
    {0,0,0,0,0,0,0,1,1,1,1} };
static int ghamming[4][7] = { {1,1,0,1,1,0,1},
    {1,0,1,1,0,1,1}, {0,1,1,1,0,0,0} , {0,0,0,0,1,1,1} };
bit_matrix h7_11::gen_matrix((int**)ghamming,4,7);
bit_matrix h7_11::p_matrix((int**)phamming,4,11);

// Base_Code::~Base_Code() { cerr << "\nBase_Code destructor called \n"; }

ascii_vector h7_11::decode(const bit_vector& codeword) const {
    ascii_vector dekode;
    dekode.lvalue(0, codeword.elem(2));
    dekode.lvalue(1, codeword.elem(4));
    dekode.lvalue(2, codeword.elem(5));
    dekode.lvalue(3, codeword.elem(6));
    dekode.lvalue(4, codeword.elem(8));
    dekode.lvalue(5, codeword.elem(9));
    dekode.lvalue(6, codeword.elem(10));
    return dekode;
}

bit_vector h7_11::closest_codeword(const bit_vector& mdv) const {
    bit_vector correct_vector = mdv.deepcopy();
    int error_bit = syndrome(mdv);
    if (error_bit == 0) {
        return  correct_vector;
    } else {
        if (error_bit <= 11) {
            correct_vector.lvalue(error_bit-1,
            mdv[error_bit-1] ^ one_mod_2);
            return correct_vector;
        } else return bit_vector::zero(mdv.size());
    }
}

int h7_11::syndrome(const bit_vector& mdv) const {
    bit_vector temp = (h7_11::p_matrix)*(mdv);
  static int pow2[] = {1,2,4,8};
  int sindrome = 0;
  for (int i = 0; i < 4; i++)
```

```
        if (temp[i] == one_mod_2) sindrome += pow2[i];
   return sindrome;
}

bit_vector h7_11::generate(const bit_vector& mv) const {
     bit_vector codeword(11);
     bit_vector temp = gen_matrix*mv;
     codeword.lvalue(0, temp[0]);
     codeword.lvalue(1, temp[1]);
     codeword.lvalue(3 , temp[2]);
     codeword.lvalue(7 , temp[3]);
     codeword.lvalue(2 , mv[0]);
     codeword.lvalue(4 , mv[1]);
     codeword.lvalue(5 , mv[2]);
     codeword.lvalue(6 , mv[3]);
     codeword.lvalue(8 , mv[4]);
     codeword.lvalue(9 , mv[5]);
     codeword.lvalue(10 , mv[6]);
     return codeword;
}

// this will generate a 7/15 codeword set

cyclic7::cyclic7(const bit_vector& poly_sr_coef, int length, int cor_dist) :
     shreg(poly_sr_coef), code_length(length), correcting_distance(cor_dist) {
     //        Tracer tr("cyclic7");
     const int CODE_SET_SIZE = 128;
     word_length = poly_sr_coef.size();
     char temp_char;
     ascii_vector temp_vector;
     code_ptr = new bit_vector[CODE_SET_SIZE];
     for (int i = 0; i < CODE_SET_SIZE; i++) {
          temp_char = i;
          temp_vector = ascii_vector(temp_char);
          shreg.init(temp_vector);
          code_ptr[i]  = shreg.generate_seq(code_length);
     }
}

cyclic7::~cyclic7() {
    delete[] code_ptr;
}

int cyclic7::correct_code(const bit_vector& mdv) const {
   const int CODE_SET_SIZE = 128;
   int temp_weight = mdv.size();
   int w;
   int index = -1;
     for (int j = 0; j < CODE_SET_SIZE; j++)    {
          if ( (w = mdv.hamming_distance(code_ptr[j]) )  <=  temp_weight)  {
```

```
            temp_weight = w;
            index = j;
        }
    }
  return index;
}

ascii_vector cyclic7::decode(const bit_vector& mdv) const {
    ascii_vector temp_vector;
    for (int i = 0; i < word_length; i++) temp_vector.lvalue(i,mdv[i]);
    return temp_vector;
}

bit_vector cyclic7::closest_codeword(const bit_vector& mdv) const {
    int k = correct_code(mdv);
    return contents_ptr()[k];
}

int cyclic7::minimum_distance() const {
    int min_distance = code_length;
    const int CODE_SET_SIZE = 128;
    for (int j = 1; j < CODE_SET_SIZE; j++)
        for (int k = j+1; k < CODE_SET_SIZE; k++) {
            int dis = code_ptr[j].hamming_distance(code_ptr[k]);
            if (dis < min_distance) min_distance = dis;
        }
    return min_distance;
}

bit_vector cyclic7::generate(const bit_vector& mdv) const {
   shreg.init(mdv);
   return (shreg.generate_seq(code_length));
}
```

10.7.2 File *m2_d7.h*

```
// file m2_d7.h
#ifndef MOD2_DIM7_H
#define MOD2_DIM7_H
#include "bitref.h"

class ascii_vector : public bit_vector {
public:
  static int global_dimension;
  static unsigned char mask[8];
  ascii_vector();
  ascii_vector(int* ptr_int);
  ascii_vector(const ascii_vector&);
  ascii_vector(char aski);
```

```
   operator char ();
   ascii_vector(const bit_vector& mv) : bit_vector(mv) {};
   ascii_vector& operator=(const ascii_vector& mv);
   ~ascii_vector() {};
};

#endif MOD2_DIM7_H
```

10.7.2 File *m2_d7.cpp*

```
// file m2_d7.cpp
#include "m2_d7.h"

int ascii_vector::global_dimension = 7;
unsigned char ascii_vector::mask[8] = {1,2,4,8,16,32,64,128};

ascii_vector::ascii_vector() : bit_vector(global_dimension) {
   // modulo_number init(0);
     for (int i = 0; i < global_dimension ; i++) {
       lvalue(i, zero_mod_2);
     }
}

ascii_vector::ascii_vector(int* ptr_int) :
     bit_vector(ptr_int, global_dimension) {}

ascii_vector::ascii_vector(const ascii_vector& mv) : bit_vector(mv) {}

ascii_vector::ascii_vector(char aski) : bit_vector(global_dimension) {
     //    modulo_number zero(0);
     //    modulo_number one(1);
     // run through the bits and test
     for (int i = 0; i < size(); i++) {
          if (mask[i] & aski) lvalue(i,one_mod_2); else lvalue(i,zero_mod_2);
     }
}

ascii_vector::operator char () {
     //    modulo_number one(1);
     unsigned char temp = '\0';
     for (int i = 0; i < 7; i++) {
          if (elem(i) == one_mod_2) temp |= mask[i];
     }
     return temp;
}

ascii_vector& ascii_vector::operator=(const ascii_vector& mv) {
     bit_vector::operator=(mv);
     return (*this);
```

```
}
```

10.7.2 File sreg7.h

```
// this is the header file for shift register class
// file sreg7.h
#ifndef SHF_REG_7_H
#define SHF_REG_7_H
#include "m2_d7.h"

class shf_reg_7 {
private:
   int size;
   bit_vector feedback;
     bit_vector state;
public:
   shf_reg_7(bit_vector feedback);
   shf_reg_7(const shf_reg_7&);
   void init(const bit_vector&) const;
     shf_reg_7 operator=(const shf_reg_7&);
     unsigned char operator()() const;
   bit_vector generate_seq(int) const;
     friend ostream& operator<<(ostream&, const shf_reg_7&);
};

#endif SHF_REG_7_H
```

10.7.2 File sreg7.cpp

```
// this implements shf_reg class
#include "sreg7.h"
#include "tracer.h"
#include "iostream.h"

shf_reg_7::shf_reg_7(bit_vector coefficients) : size(coefficients.size()),
     feedback(size) , state(size) {
     feedback = coefficients.deepcopy();
     state = bit_vector::zero(size);
     //  cout << "feedback is "  <<  feedback;
}

ostream& operator<<(ostream& s, const shf_reg_7& sr) {
     s << "\n";
     s << "feedback is " << sr.feedback;
     s << "\nstate is " << sr.state;
     s << "\nsize is " << sr.size << "\n";
     return s;
}
```

```
shf_reg_7::shf_reg_7(const shf_reg_7& sr) {
     size = sr.feedback.size();
     feedback = sr.feedback;
     state = sr.state;
}

void shf_reg_7::init(const bit_vector& init_state) const {
     if (init_state.size() != size) bit_array::error("bad shift register init");
     for (int i = 0; i < size; i++) {
          if (init_state[i]) state.lvalue(i,one_mod_2);
          else state.lvalue(i, zero_mod_2);
     }
}

shf_reg_7 shf_reg_7::operator=(const shf_reg_7& sr) {
     size = sr.feedback.size();
     feedback = sr.feedback;
     state = sr.state;
     return *this;
}

unsigned char shf_reg_7::operator()() const {
//   Tracer tr1("shf_reg_7::op()");
   // one
     unsigned char mn = state[0];
   unsigned char temp = zero_mod_2;
     for (int i = 0; i < size; i++) {
          if (state[i] == one_mod_2 && feedback[i] == one_mod_2)
               temp ^= one_mod_2;
     }
     for (int j = 0; j < size-1; j++) {
          //      state[j] =state[j+1];
          state.lvalue(j, state[j+1]);
   }
     // state[size-1] = temp;
     state.lvalue(size-1,temp);
     return mn;
}

bit_vector shf_reg_7::generate_seq(int length) const {
     bit_vector mdv(length);
     for (int i=0; i< length; i++) {
          //     mdv[i] = (*this)();
      mdv.lvalue(i, (*this)() );
   }
   return mdv;
}
```