CHAPTER 9
BIT ARRAYS

9.1 Introduction

The language C comes with bit arrays of predefined length. These are the integral types **char**, **int**, **long**, etc., with the bitwise operators **&**, ^, <<, etc. There are two problems with these: (1) only arrays of the exact length 8, 16, and 32, etc., are provided and (2) there is only the minimal functionality for these. We want to have arrays of all lengths and dimensions, plus more functionality. Bit arrays make up a beautiful area for an object-oriented design in C++.

Bit arrays are nothing more than mathematical arrays of integers restricted to the set {0,1}. They can be used to represent sets, in particular subsets of the ASCII character set, pixel patterns, and modulo two vectors. We could use the vector template implementation of a previous chapter instantiated with modulo two numbers to get the basic storage requirements and functionality implemented. Then we could derive from this a modulo two vector class to obtain a character set class and do another derivation in a different direction to obtain some added functionality for modulo two vectors.

It is a better idea to design the bit array classes from the ground up. What was the path of evolution for the bit array code presented in this chapter? First, a non-template implementation of **rational_vectors** was developed. Then an implementation of **rational_matrix**, as a private derivation of **rational_vector**, followed. After this, a **modulo_number**, **modulo_vector**, and **modulo_matrix** implementation was produced by editing the respective rational number classes. Modulo two vectors wasted a lot of space because a full byte was used to store a **0** or a **1**. Furthermore, this implementation did not take proper advantage of the built-in bitwise operators on integral types, **chars** in particular.

There was a hidden agenda for the development of modulo two vectors: "Could a neural net recognize patterns better by using algebraic coding theory?". An implementation of modulo two vectors was needed; slow and clumsy ones would be fine to test the theory. Later, the slow and clumsy ones could be replaced by fast and agile ones.

Beside the modulo two vector implementation, a character set implementation was needed for validation of keyboard input. Back in the days of UCSD Pascal, a method of employing subsets of the ASCII character set for keyboard filtering (as each character was entered) of input such as (1) integers, (2) dollars and cents, (3) floating point numbers, and (4) polynomials proved useful in designing user interfaces. Essentially, these are examples of what we call **regular expressions**. The problem would have been of a pure finite state machine, except we have the backspace character. This changed the problem to that of a **stack machine pushdown automata (PDA)** that was close to the underlying FSM for the regular expression. Each **state** needed a *valid* set of characters in order to make a transition. Pascal has a **set** implementation as part of the language, so this worked well. C supplanted Pascal in popularity, but there was no set implementation in C. Thus, this had to be done with a clumsy use of pointers. But with C++, the whole picture changed. The ugly C implementation could be made beautiful in C++.

We are going after two distinct implementations here: (1) modulo two vectors with their modulo two addition and scalar multiplication and (2) ASCII character sets. This implementation was an example of factoring out basic data storage and functionality into a common base class, always a good idea as long as our derivation hierarchy does not get too deep. Cosmetically factoring out for only a few members in a large design can create too many levels of inheritance;

this can lose the user. Three levels or less is comprehensible.

9.2 The Common Base Class

The common base class will be named **bit_array**. Much of the style used parallels that of previous implementations of arrays. We will continue to use the handle-body idiom in the usual form. The underlying C array for this implementation will be of an array of **unsigned chars**. It should be noted that unsigned **ints** or **unsigned longs** could have (or maybe should have) been used. If the address bus passed around 4 bytes as fast as 1 byte and the processor handled 4 bytes as fast as 1 byte, then a 4 byte implementation would have been better, though less readable. (See Shapiro[] for a basic implementation based on the size of a **long**.). One byte was chosen for the available simplicity.

The data member part of class **bit_array** is:

```
class bit_array {
private:
    struct bit_array_rep  {
        int  dimension;
        int  no_bytes;
        int  extra_bits;
        int refs;
        unsigned char* ptr;
    }*  rep;
    ...
    ...
    ...
};
```

First of all, the number of bytes needed to store the bit components of the array is not the actual size of the array because we store eight bits in a byte. We obviously need **dimension** as a data member. The number of bytes needed is the smallest number of bytes that will hold **dimension** number of bits. For example, an eleven bit vector needs two bytes with eight bits stored in the beginning byte and the three extra bits in the last byte. Now, all this can be calculated from the **dimension** using the formulas: (1) **no_bytes = (dimension-1)/8 + 1** and **extra_bits = dimension%8**. Since we always need these values, we make them data members of the **bit_array_rep**. This eliminates repeated calculations, and costs us nothing since only pointers to **reps** are copied, not the actual **reps**. We use the handle-body idiom because the ASCII character sets use sixteen bytes (128 bits) and the powerful shift register sequence also uses sixteen bytes (127 bits), and we don't want to deepcopy this large amount of bytes.

9.2.1 Constructors for bit_array

The basic constructor for **bit_array** takes an **int** as an argument, allocates a **bit_array_rep**, allocates bytes to hold the bit components, and initializes these bits to **0**.

A default constructor for **bit_array** is needed to create dynamic arrays of **bit_arrays**

themselves using **new**. The statement **new bit_array[n]** calls the default constructor for **bit_array n** times. See section 9.6.3 for a discussion on the use of the default **bit_array** constructor as part of the **bit_matrix** constructor. The default constructor for **bit_array** creates a **bit_array_rep** and assigns zero values to all **rep** data members including the deep memory pointer. The member function **void bit_array::assign(int dim, unsigned char* uptr)** will be used to attach valid memory to these empty arrays and fill in the data members of **rep** when the time comes to initialize a **bit_array**.

The copy-initializer performs the usual tasks. It increments the **ref** count on the object being copied and then assigns the **rep** pointer of the copied object to the **rep** pointer of the object being created.

9.2.2 Destructor for bit_array

The code body for the **bit_array** destructor is straightforward. It is:

```
bit_array::~bit_array()    {
    if  (--rep->refs == 0)  {
        delete[]  rep->ptr;
        delete  rep;
    }
}
```

If the **refs** count drops to zero, **delete[] rep->ptr** will free the array of **chars** created by **new[]** at the address **rep->ptr**. The **[]** needs no argument; the heap table knows how many bytes were allocated at this address. Then **delete rep** will free the memory to which **rep** pointed. Notice we don't delete **rep** first because we need access to **rep->ptr**. We must go to the deepest memory and then back out.

9.2.3 The Assignment Operator for bit_array

As always the assignment operator is essentially a sequential combination of a destructor like call on the calling object and a copy-initializer like call to reconstruct the calling object. The *copy-into* part looks similar to the code of the copy-initializer. The code body of the assignment operator is:

```
bit_array::operator=(const bit_array& ba)  {
    if  (rep->ptr  != ba.rep->ptr)  {
        if    (--rep->refs  == 0)  {
            delete[]  rep->ptr;
            delete  rep;
        }
        rep  =  ba.rep;
        rep->refs++;
    }
}
```

If the **reps** are pointing to the same deep memory of components, this is self-assignment. We don't want to deallocate any memory of the calling object when the calling object and the argument are identical. Then we lose the deep memory of the argument. For this reason we have the **operator!=** check on the **rep** ptrs. Always do this check when deep memory is involved in the assignment operator.

9.2.4 Bit Access Member Functions

There are two useful functions that have been put in the **protected** category of the class definition because they give the user direct access to the memory locations where the array representation is stored. They are (1) **unsigned char* address_array() const** which returns the address of the deep memory that holds the bytes that store the bit components, and (2) **struct bit_array_rep* address_array_rep() const** which returns the address of the **bit_array_rep**. Because of the **protected** designation on these, any **public** or **protected** derivation chain can access these as well as one level of **private** derivation. The **protected** designation keeps it from being used by functions that are not member or **friend** functions in any derivation hierarchy.

9.2.5 Bit Masking Arrays

In order to do the bit work efficiently and elegantly, some constant bytes and arrays of bytes are used. In this implementation, at the top of the header file are declared some **static const chars** and **static const char** arrays. They are not embedded in the class definition; maybe they should be. This little bit of cleanup will be left as an exercise. These variables are stored globally and are **private** to the file because of their C **static** definition. Only the code in the file that they are compiled within can use them. Furthermore, they cannot get inserted more than once because the header file has the usual **#ifndef...** lock that prevents reinsertion. When including **bitset.h**, do not attempt to change these **static** variables in the code files into which they have ben included. They have been declared **const** with this in mind. **const** means they have to be initialized at the time of definition and after that cannot be changed.

It is not a good idea to imbed constants within code. In C, we commonly defined constants by putting statements such as **#define zero_mod_2 0** and **#define one_mod_2 1** near the top of a source file; C++ style does not rely on the preprocessor for this embedded constant problem, but uses **const**. Therefore, we see **static const char zero_mod_2 = 0** instead of **#define zero_mod_2 0** .

Conveniently we can initialize **unsigned chars** using literal integers. The array **bit_check** creates an array of bytes in which exactly one bit is set to one in each byte:

(1) **bit_check[i]** has the it bit set to one where 0 <= i < 7;
(2) **bit_mask** has leading bits (low-order) in each byte set to one;
(3) **bit_mask[0]** has all eight bits set to one;
(4) **bit_mask[1]** has the zeroth bit set to one;
(5) **bit_mask[7]** has the leading seven bits set to one;
(6) **front_end_mask** has the high-order bits set to one;
(7) **front_end_mask[7]** has the seventh bit set to one;

(8) **front_end_mask[6]** has bits 6 and 7 set to one;
(9) **front_end_mask[5]** has bits 5, 6, and 7 set to one.

These are used in the implementation of various member functions.

9.2.6 Overloaded Bitwise Operators

There is not too much surprising at the **bit_array** level. We use the bit operators that come with the C language to accomplish our overloading of them for **bit_arrays**. Let's review what each symbol essentially means at the bit level. The C function works on the respective bits in the case of the binary operators, and the overloaded C++ functions merely calls the respective C function for each byte in the deep memory. At the bit level we have the following table of logic rules:

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	0	1
0	1	0

Figure 9.2.6

The **operator&** and the **operator|** are intended to be used for the intersection and union of sets respectively. The **operator^** is used for modulo two addition where 1 + 1 = 0. These base class operators are being defined taking into account the **bit_set** and **bit_vector** classes that will be derived from **bit_array**.

9.2.7 Bit Utilities

We have to be able to test a bit, set a bit to one, and clear a bit to zero. The integer divide, /, and remainder operator, **%**, are useful for this. If we want the **n**th bit, we look at byte numbered **n/8** in the array and bit numbered **n%8** in this byte. Then we use **bit_check[n%8]** with **rep->ptr[n/8]** to accomplish the task. The order that we defined the static arrays of bytes was selected to work in conjunction with this scheme. The member functions **test_bit**, **set_bit**, and **clear_bit** take care of these low-level bit operations.

We cannot use the bracket operator traditionally. Remember the bracket operator usually returned a reference to the location where a variable was actually stored; this allowed us to use **[]** on the left of **operator =** to assign into a component or on the right to access a component. In this case, a bit is not being stored in a manner in which we can get a *narrow* enough reference. We will see a problem with the bracket operator in the chapter on polymorphic matrices also. We can have **operator[]** return a value instead of a reference. It can't return a bit, so it returns either a **zero_mod_2** or **one_mod_2** byte. The function **char elem(int),** which invokes **[],** has the same functionality as **operator[]**. To accomplish the *lvalue* functionality usually accomplished by **[]** or **elem**, a more awkward member function could be defined. Its definition would be:

```
void lvalue(int i, unsigned char uch) const {
    if (uch & one_mod_2) set_bit(i); else clear_bit(i);
}
```

The standard functionality was accomplished by using an operator[] with proxies. See the code for details and consult the book "more Effective C++" by Scott Meyers.

9.2.8 Hamming Weight

The Hamming weight of a modulo two vector is the number of times that a **1** occurs as a component of that vector. This trivial concept is used to calculate a distance between two modulo two vectors. The concept of distance between modulo two vectors is the fundamental building block of algebraic coding theory. The distance between two vectors of the same size is defined to be the number of bit locations in which they differ. This can be obtained by adding the two mod 2 vectors and counting the number of **1**'s in the summand. This yields the formula: **distance(v1, v2) = hamming_weight(v1 + v2)**. Why is distance important.? For a complete set of vectors of some like dimension, we will select a special subset and designate it as a set of *codewords*. We would like to have this set of codewords distributed in some systematic way among the entire set of vectors in order that each vector in the larger set can be associated with exactly one codeword. This will be a many-to-one mapping. How is this mapping determined? For each vector, find the codeword closest to it using the above distance formula. It should be noted that much sophisticated mathematics has been developed for the choice of codeword sets and calculations with them.

Obviously, we need to calculate the hamming weight in the most efficient way as possible. To calculate the weight of a modulo two vector, we will sum up the weights of the bytes representing the array. This reduces down to initially calculating the weights of all 256 possible bit-patterns that a byte can assume. Since we do not want to calculate these weights thousands of times during a program execution; we should immediately build a *look-up* table the first time we need to count the bits in some byte and then use this table in all future requests. We could hard code this table into an array, but that is too much source code editing for this presentation. Let's look at the code of the **static** member function **bit_count**:

```
int bit_array::bit_count(unsigned char uch) {
    static int count[256];
    static char init = 1;
```

```
    int  temp_count;
    if  (init)  {
        for  (int  j  =  0;  j  <  256;  j++)  {
            temp_count  =  0;
            unsigned  char  jj  =  (unsigned  char)j;
            for  (int k = 0;  k < 8;  k++)  if  (jj  &  bit_check[k])  temp_count++;
            count[j]  =  temp_count;
        }
        init = 0;
    }
    return  count[uch];
}
```

The above code illustrates the concept of a C **static** variable being declared inside a function body. A **static** variable of any kind is not stored on the stack; it is stored in a global data area. The values assigned into **statics** remain intact from exit and reentry into a function. The following describes how the statics work in **bit_count**. Upon first entry into **bit_count**, the statement **static char init = 1** assigns the value one into **init**. This statement is not executed upon reentry; **init** will maintain the value it had at the previous exit. During the first entry, **init** being equal to one forces execution of a block that assigns values to an array of 256 elements, and **init** is assigned the value zero. For ever on during this execution, **init** will be zero and we will not recompute the **static** array **count**. **count[i]** will be the number of bits set to **1** in the byte whose binary value is **i**. This works nicely, and is a beautiful use of **static** C variables. Another use of static variables could be for a random number generator.

To calculate the Hamming weight, we mask out the unused bits of the last byte and loop through all the bytes, summing up the values returned by **bit_count**. The convenient aspect of this approach is that we use the binary value of a byte, when viewed as an **unsigned char,** to index ourselves into this array of Hamming weights.

9.2.9 Header File for bit_array

```
// class definitions for bit_array, ASCII_set, and bit_set
#ifndef BIT_CLASSES_AND_BITSET
#define BIT_CLASSES_AND_BITSET
#include <iostream.h>
static const unsigned char zero_mod_2 = 0;
static const unsigned char one_mod_2 = 1;
static const unsigned char bit_check[] = {1,2,4,8,16,32,64,128};
static const unsigned char bit_mask[] = {255,1,3,7,15,31,63,127};
static const unsigned char front_end_mask[] = {255,254,252,248,240,224,192,128};
class bit_array {
private:
    struct bit_array_rep {
        int dimension;
        int no_bytes;
        int extra_bits;
        int refs;
```

```
        unsigned char* ptr;
    }* rep;
protected:
    unsigned char* address_array() const { return rep->ptr; }
    struct bit_array_rep* address_array_rep() const { return rep; }
public:
        class BitProxy {
        public:
                BitProxy(bit_array& ba, int position);
                BitProxy& operator=(const BitProxy& rhs);
                BitProxy& operator=(unsigned char m2);
                operator unsigned char() const;
        private:
                bit_array& the_bit_array;
                int bit_position;
        };

public:
    void assign(int , unsigned char*);
    int size() const { return rep->dimension; }
    int byte_size() const { return rep->no_bytes; }
    bit_array(int); // constructor
    bit_array();
    bit_array(const bit_array&);  // copy-initializer
    bit_array& operator=(const bit_array&);  // assignment equals
    bit_array() {
        if (--rep->refs == 0) {
            delete[] rep->ptr;
            delete rep;
        }
    }  //destructor
    bit_array operator&(const bit_array&) const; // and operator
    bit_array operator|(const bit_array&) const; // or operator
    bit_array operator^(const bit_array&) const; // exclusive or operator
    bit_array operator() const; // complement operator
    int operator==(const bit_array&) const; // logical equals
    int operator!=(const bit_array&) const; // logical not equals
    int test_bit(int n) const { return (rep->ptr[n/8] & bit_check[n%8]);}
    void set_bit(int n) const  { rep->ptr[n/8] = rep->ptr[n/8] | bit_check[n%8];}
    void clear_bit(int n) const  { rep->ptr[n/8] = rep->ptr[n/8] & (bit_check[n%8]);}
    static void error(char*);
    unsigned char operator[](int i) const {  if (test_bit(i) ) return one_mod_2;
        else return zero_mod_2; }
        unsigned char elem(int i) const { return (*this)[i];
    }
    static int bit_count(unsigned char uch) ;
    int weight( ) const;
    bit_array deepcopy() const;
```

```
    const BitProxy operator[](int position) const;
    BitProxy operator[] (int position);
    friend class BitProxy;

    friend ostream& operator<<(ostream&, const bit_array&);
    friend istream& operator>>(istream&, bit_array&);
};
```

9.3 The Classes bit_set and ASCII_set - Only Added Functionality

9.3.1 Reusing All Accessible Member Functions

The classes **bit_set and ASCII_set** are examples of derivation with added functionality but no new added data members. We had another example of this kind of derivation in Chapter 6 with the **ad_matrix** class. We will again discuss what this means as far as inheriting member functions from the base class. Because **bit_set** is a publicly derived class of **bit_array**, the **protected** and **public** members of **bit_array** remain **protected** and **public** respectively in **bit_set and ASCII_set**. **ASCII_set** is a publicly derived class of **bit_array** with a fixed dimension of 128 bits, 16 bytes. Any discussion of **bit_set** will pertain to **ASCII_set** as well. Public derivation was used because a **bit_set** is a **bit_array** with added functionality. An *is a* relationship dictates public derivation. In the Chapter 5 discussion of private derivation of a matrix off of a vector, we had an example of private derivation for reuse of the vector implementation. A matrix is not that close to being a vector, but a **bit_set** and **bit_array** are real close. This **private** or **public** decision is not always clear.

First of all, an object of a derived class can be the calling object of a base class member function that is **public** or **protected**. This is always true whether the derived class has new data members or not. Essentially, the base class slice of the object of the derived class does the calling. This is not a problem. The second consideration is for any function having a formal parameter, reference or not, of a base class object. Here too, this function can be called with an argument that is an object of the derived class; in the code body only the base part of the object will be used. There is no problem yet. The third consideration is return value. The following statement applies to class derivations with new data or not: if the return type of the base member function is not of the base type or a type higher up in a derivation hierarchy, then that member function is fully functional for the derived class. An example of this would be the Hamming weight function, **weight**, of **bit_array** that returns an **int.** The function **size** is another such example. Where does inheritance fail?

The binary arithmetic operators are examples of member functions that return a base type object. In this implementation, the overloaded bitwise operators like **&** do this. If **a** and **b** were two **bit_sets**, then **a & b** would return a **bit_array**. We want a **bit_set** though. If a **bit_set** had added data members, we would have a problem; **operator&** would not be returning enough information. Because **bit_set** has no new data members, we can easily convert a returned **bit_array** into a **bit_set**. First we need to define an **operator=** with implementation:

```
bit_set& bit_set::operator=(const bit_array& ba) {
    bit_array::operator=(ba);
```

```
    return (*this);
}
```

What is actually happening here? We have the hidden pointer **this** calling **bit_array::operator=(ba)**. This is fine; a **bit_set** can call a **bit_array** member function. What happens is that **ba** is assigned into the **bit_array** part of ***this**, a **bit_set.** But this is all the data of ***this**, and thus the assignment is complete. This redoes **operator=** for a **bit_array** argument, but it was cosmetic and reused the base **operator=**.

This version of **operator=** is used in the following code segment:

```
main()  {
    bit_set  a,b,c;
    ...
    c  =  b &  a;
}
```

Here, **bit_array::operator&(a,b)** with two **bit_set** arguments returns a **bit_array**, but the above special **operator=** assigns the **bit_array** into a **bit_set**. This solves reusing base member functions that return a base object in which there is a derivation of no new data members.

What about the sister function of **operator=**? We also need a special kind of copy constructor:

```
bit_set::bit_set(const  bit_array&  ba)  :  bit_array(ba)  {}
```

As one would expect, its base constructor call is merely to the **bit_array** copy-initializer. That takes of all the data, done! This could be used to convert a **bit_array** to a **bit_set** directly or indirectly by implicit conversion. Let's look at the following code:

```
main() {
    bit_set  a,b,c,d;
    ...
    d  =  a  *  ( b  &  c);
}
```

Realize that a derived class argument can be passed to a base class formal parameter, but in general a base class argument cannot be passed to a derived class formal parameter. The base class argument must be converted to a derived class object. Since there is no new data, we can write a derived class constructor that will make this conversion when necessary. In the above code segment, **operator*** (for intersection) of **bit_set** needs a **bit_set** as an argument, but **b & c** is a **bit_array**. This special constructor will automatically convert **b & c** to the argument type **operator*** needs, **bit_set**. Then we are in business.

This **operator=** and this special copy constructor overlap in their duties. With this copy constructor, we should not need the special **operator=** for the statement **c = a & b**. We could simply use the standard **bit_set& bit_set::operator=(const bit_set&)**. With this standard **operator=**, the statement **c = a & b** would be interpreted as: **c.bit_set::operator= (a.bit_array:: operator&(b))** and the return value of **a & b** would automatically be converted to a **bit_set** by

this special copy constructor.

Automatic conversions , assuming one wrote proper constructors and conversion operators, only take place one level deep. Sometimes one has to explicitly call for the conversion; this makes the code clearer even if not necessary. We could have explicitly written **c = bit_set(a&b)**. The same functions are called; it is only explicit here.

All of this was explained in the greatest detail because derivation for only added functionality is useful, and we get to reuse the base class functionality with at most a little extra syntax provided by these two special sister functions.

9.3.2 Class definition for ASCII_set

```
class ASCII_set : public bit_array {
public:
    ASCII_set() : bit_array(128) {}
    ASCII_set(char* s) ;
    ASCII_set(const ASCII_set& bs) : bit_array(bs) { } //copy-initializer
    ASCII_set(const bit_array& ba) : bit_array(ba)  {
        if (ba.size() != 128)
        error("\nIn ASCII_set(ba) ba does not have dimension 128\n");
    }
    ASCII_set& operator=(const ASCII_set& bs)  {
        bit_array::operator=(bs);
        return (*this);
    }
    ASCII_set& operator=(const bit_array& ba)  {
        if (ba.size() != 128)    error("\nIn ASCII_set(ba) ba does not have dimension 128\n");
        bit_array::operator=(ba);
        return (*this);
    }
    ASCII_set() { }
    int test_char(unsigned char ch) { return test_bit(ch); }
    void remove_char(unsigned char ch) { clear_bit(ch); }
    void insert_char(unsigned char ch) { set_bit(ch); }
    ASCII_set operator+(ASCII_set& bs) { return *this | bs; } // set union
    ASCII_set operator-(ASCII_set& bs) { return *this ^ (*this * bs); }
    ASCII_set operator*(ASCII_set& bs) { return *this & bs;} // set intersection
    char filter_char();
    ASCII_set set_minus_string(char*);
    ASCII_set set_plus_string(char*);
    static int in_string(char,char*);
    friend ostream& operator<<(ostream& s,ASCII_set& bs);
    friend istream& operator>>(istream& s,ASCII_set& bs);
};
```

9.3.3 Bit Mapping of the ASCII Character Set

The intended use for **ASCII_set** is to model subsets of the ASCII character set. There are

128 ASCII characters, and we will bit-map subsets into sixteen bytes (128 bits). An 128 **bit_array** will contain ASCII character number **i** exactly when the **i**th bit is set to **1**. If the **i**th bit is **0**, the ASCII **char i** will not be in the set. For example, the set {**'A'**, **'a'**} will have bits 65 and 97 set to **1**; all others will be **0**. A set of digits will have bits 48 through 57 set to **1**, all others will be **0**. As stated above, the proper byte and bit numbers in the deep memory making up the set are located using integer divide and the **%** operator for **ints**.

An easy way to initialize a set is to provide a constructor that builds a set from a C string. For example, we see the following:

```
ASCII_set DIGITS("0123456789");
ASCII_set SIGNS   = "+-";
```

9.3.4 Set Operations

It turns out that we can overload the operators **+**, *****, and **~** for union, intersection, and complement, respectively. The precedence, associativity, commutativity, and distribution laws of ***** and **+** correspond with those of intersection and union. The **operator-** has been overloaded for set subtraction, but it does not mathematically associate correctly. Don't put **-** in a mixed expression unless you know exactly what you are doing.

9.4 A Keyboard Filtering Application

Often a program requires the entry of a string of characters that satisfies a certain format. This includes (1) integer format, (2) floating point format, (3) dollars and cents format, (4) polynomial format, etc. Each of these different formats here can be modeled by a finite state machine. A finite state machine will validate a string. A string of characters is processed from left to right, and after processing a character we enter a new state. In this state, there is an acceptable set of characters into which the next character must belong. If and when the string is completely processed, we accept it as valid if we are in an accepting state. In Figure 9.4, we have a finite state machine for strings with dollars and cents format. **S** is the starting state and **T** is an accepting state. (For further reference, consult a discrete math or automata book that discusses FSMs.)

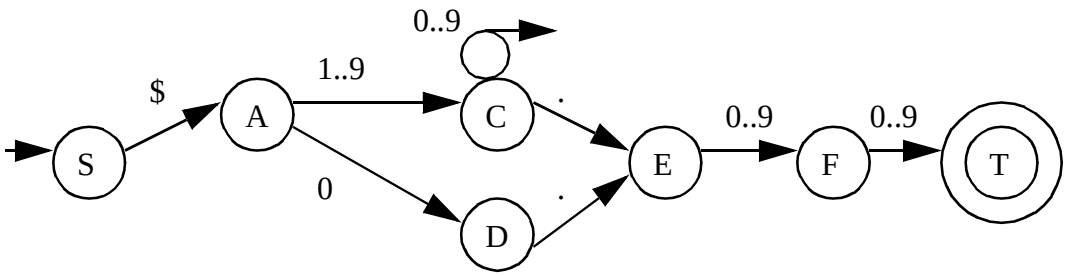

Figure 9.4

Normally, the validity of a string is checked after it is completely assembled. For keyboard input (if the terminal is in the right mode on Unix machines), we could immediately check to see if a character is *acceptable* to be appended to the current string. For example, in the above FSM, after we enter a **$**, we must follow it by a digit. If that digit is **0**, we must follow it by a **.** . At each state, we have a group of states we can transition to, and for each transition we have a set of acceptable characters that lead us from one state to another. So, for an arc in a FSM, we have a set that goes with it. This is why a class that models subsets of the ASCII character set is needed. If a character is entered at the keyboard that causes no transition, then we do not want the input function of the program to add this character to the current string, but to beep and then erase the character on the screen. This will signal the user to enter an acceptable character. There is one problem with modeling the keyboard input with FSMs; it is that we must allow for backspace. The backspace character has to be allowed as an input. It is not considered as a character in the string, but rather as erasing the last character in the string, backing up to a previous state. In the case of the dollars and cents FSM, if we erase a period, we don't know whether to back up to state C or state D. The solution for this is to create an auxiliary stack and store the new state on the stack after each transition. If we key in a backspace, we remove the last character from the string and pop the last state off the stack. Then we are now put into the state on top of the stack. For this string entry process, the carriage return will send us into an accepting state. This algorithm is an example of a *pushdown automata* algorithm. The significant accomplishment is that we have taken a regular language consisting of a set of strings recognized by a FSM, added the *return* and the *backspace* to be used in conjunction with a keyboard and dynamic validation, and then constructed a related pushdown automata to complete the model. It should be mentioned that a return is only accepted in a state where up to this point a valid string has been entered. In the above example, **T** will be the only state from which we can enter a return, and that will move us into a state **DONE** from which we cannot process anymore characters. After running through, from left to right, the string of characters including backspaces entered at the keyboard, and applying the backspace operator to the preceding character, a string accepted by the original machine will be left. The dollar and cents keyboard entry algorithm is left as an exercise.

The crucial function of **ASCII_set** that does this filtering is **filter_char**:

```
char ASCII_set::filter_char() {
    char ch;
    while (test_char(ch = getch()) == 0) putchar('\7');
    if ( (ch>=' ') && (ch<='') ) putchar(ch);
    return ch;
}
```

We call **filter_char** with the **ASCII_set** calling object that is the full set of characters that will allow us to make a transition from the current state. This calling object is the union of all the **ASCII_sets** on arcs leading out of the current state in the FSM, plus backspace and return if they are acceptable.

The following function named **intread** does keyboard filtering for strings that represent an integer in the range from -32768 to 32767. It has extra functionality to temporarily accept an integer out of this range, and then go through and erase it. It is not written in a general manner

that reflects that there is a FSM based PDA in the background. Beautification is left as an exercise. Here is **intread**:

```
#include "bitsets.h"
#include <string.h>
#include <stdio.h>
#define SETSIZE 16
intread() {
    int count,flag1,i,j,max,number,posflag,signflag;
    char *w,*ptr,ch,
    instring[10];
    int maxlen = 6;
    ASCII_set DIGITS("0123456789");
    ASCII_set SIGNS("+-");
    ASCII_set BSRET("\10\15");
    ASCII_set BSDIGITS = DIGITS.set_plus_string("\b");
    ASCII_set BSPRETDIGITS = BSRET + DIGITS;
    ASCII_set DIGITS_AND_SIGNS = DIGITS + SIGNS;
    w = instring; /* initialize to instring */
    /* flag1 = 0 means the string is acceptable for a 2-byte signed integer */
    do   {
        flag1 = 1;
        count = 0 ; /*  no chars entered yet */
        do   {
            if (count == 0)  {
                /* if a "+" sign or no sign is entered conditions are set as below */
                max = maxlen - 1;
                signflag = 0;
                posflag = 1;
                ch = DIGITS_AND_SIGNS.filter_char();
                if (SIGNS.test_char(ch))   /* tests for "+","-" */      {
                    if (ch == '-') posflag = 0; /* negative entry */
                    max = max + 1;       /* distinguishes from + */
                    signflag = 1;  /*  sign flag == 1 means + or - entered */
                }
                count++;          /* acceptable character count */
            }
            else   {
                if (count == max) /* allow bkspces/cr's in okset */    {
                    ch = BSRET.filter_char();
                    count++;
                }  else
                    if ( (count == 1) && (signflag) )  {
                        ch = BSDIGITS.filter_char();
                        count++;
                    }   else   {
                        ch = BSPRETDIGITS.filter_char();
                        count++;
                    }
```

```
            }
            if (DIGITS_AND_SIGNS.test_char(ch)) *w++ = ch; /* put in array */
            else if (ch == '\10')    /* if backspace */    {
                printf("\b%c\b",' ');              /* clean up */
                count -= 2;
                w -= 1;
            }
        } while (ch != '\15');              /* so long as no CR */
        *w++ = '\0';                        /* add '\0' at end  */
        w -= count; /* address to string that contains the  number  */
        if (signflag) ptr = w + 1; else ptr = w; /* - had one more entry */
        if ( ( (signflag) && (count < 7) ) || ( (signflag == 0) &&
            (count < 6 ) ) ) flag1 = 0;
        else /* tests if within -32768 <--> +32767 limits */
            flag1 =  0 < strcmp(ptr, posflag ? "32767" : "32768");
        if (flag1) for (j=1;j < count ;j++) printf("\b%c\b\7",' ');
    } while (flag1); /* change made on 6-5-87 */
    number = 0;
    while  (ch = *ptr++)  number = number*10 + ch - '0';
    if (posflag == 0) number = -number;
    return(number);
}
int main()   /* used for test purposes */ {
    int value;
    char ion,ch;
    char *prompt,*question;
    int i;
    /* the following tests some of the set functions */
    ASCII_set tset1("+-0123456789");
    ASCII_set tset2("+-");
    ASCII_set tset3("?!");
    ASCII_set tset4 = tset2*tset1;
    ASCII_set tset5 = tset3+tset2;
    ASCII_set tset6 = tset2-tset1;
    printf("\n");
    /* display results */
    for (i=0;i<128;i++)  if (tset4.test_char(i)) putchar(i);
    printf("\n");
    for (i=0;i<128;i++)  if (tset5.test_char(i)) putchar(i);
    printf("\n");
    for (i=0;i<128;i++)  if (tset6.test_char(i)) putchar(i);
    /* this completes the test of the above */
    /* the following tests intread() */
    ASCII_set yesno("yYnN");
    ASCII_set yes("yY");
    prompt = "please enter an integer ";
    question = "do you want to do it again? y/n ";
    do    {
        printf("\n%s",prompt);
        value = intread();
```

```
        printf("\n%d",value);
        printf("\n%s",question);
        ch = yesno.filter_char();
    } while (yes.test_char(ch));
    printf("\n");
    return 1;
    /* end of intread() test */
}
```

The function intread was originally implemented in C in which an **ASCII_set** was modeled by a **char** pointer to sixteen bytes, with no thought about a FSM based PDA. The C++ implementation of a set really cleaned up the set operations. This model could be furthered cleaned up by making a C++ class representing a directed graph and making this part of a class representing a FSM, and then extending it to the keyboard PDA.

9.5 Class bit_vector

The **bit_vector** class will be derived publicly from **bit_array** in a direction of functionality different from that for **bit_set** and **ASCII_set**. The **bit_vector** class is for vectors of modulo two numbers; it cannot be reworked for any other modulo number system such as modulo three. The reason is that one byte represents a modulo two vector of size eight, with a component being stored in a bit. If the bit is set to one, that component of the vector is one modulo two; if the bit is cleared to zero, that component is zero modulo two. Unlike, the modulo vectors of an earlier chapter, this implementation does not use a class of modulo numbers as a parameter for a vector template. If a modulo two number has to be returned, the null byte or the byte with binary value **1** will be returned. This bit-mapped implementation of modulo two vectors is much faster than the approach of an earlier chapter where an entire byte was used to store one component; here components are processed in parallel. If the **bit_array** had been implemented as an array of **unsigned ints** or **longs**, it would be faster still. This has been left as an exercise. In a later chapter, modulo two vectors are part of an algebraic coding theory package that is used to improve the pattern recognition capability of a neural net. These bit-mapped vectors provided at least a tenfold speedup in the entire neural-net algorithm. Furthermore, the need for storage dropped significantly, and this is significant within a PC-DOS environment.

9.5.1 Class Definition for bit_vector

The following is the class definition for **bit_vector**:

```
class bit_vector : public bit_array  {
public:
    bit_vector() : bit_array() {}
    bit_vector(int n) : bit_array(n) {}
    bit_vector(int *, int);
    bit_vector(const bit_vector& mv) : bit_array(mv) { }
    bit_vector(const bit_array& ba) : bit_array(ba) {}
    bit_vector& operator=(const bit_vector& mv)  {
        bit_array::operator=(mv);
```

```
        return (*this);
    }
    bit_vector& operator=(const bit_array& ba)   {
        bit_array::operator=(ba);
        return (*this);
    }
    bit_vector() { }
    bit_vector deepcopy() const;
   static bit_vector zero(int);
    bit_vector operator+(const bit_vector& mv) const  { return *this ^ mv; }
    unsigned char operator*(const bit_vector& mv) const;
    bit_vector operator*(unsigned char) const;
    bit_vector& operator+=(const bit_vector&);
    bit_vector& operator*=(unsigned char);
    int hamming_distance(const bit_vector& mv) const  { return (*this + mv).weight();}
    float* mod2_to_float() const;
    friend bit_vector operator*(unsigned char,const bit_vector&);
    friend ostream& operator<<(ostream&,const bit_vector&);
};
```

9.5.2 The Copy-Initializer and operator=

The **bit_vector** class is another example of derivation with added functionality and no added data members. The discussion of the derivation of the **bit_set** class from the **bit_array** class applies here. In that derivation, all the reference counting is handled by the base class part of the object. The copy initializer and its related copier call the base class copy initializer while both **operator=**s call the base class **operator=**. Pleasantly the reference counting is transparent to the implementation of the derived class. Regarding reference counting, see the **BigInt** or vector-matrix implementations of previous chapters.

9.5.3 Vector Arithmetic Operations

Vector addition merely uses the overloaded exclusive or **operator^** of **bit_array**; this is why it is executes fast. Eight bits are handled in parallel with a single machine code instruction, and no integer addition. Scalar multiplication uses one byte for the scalar, checks its zeroth bit, and returns either a copy of the vector operand or the zero vector. Easy!

9.5.4 Other Functionality

The destructors for both derived classes of **bit_array** have no added functionality; they automatically call the base class destructor which does the reference counting. Conveniently, the derived class destructors don't have to be in our class definitions since the code bodies are empty; the compiler automatically constructs the default destructors.

For use with algebraic coding theory, there is a function named **hamming_distance**. For the neural net application, **mod2_to_float** returns a pointer to an array of **floats** that represents

modulo two numbers with a floating point equivalent. In the neural net, zero modulo two becomes **0.1** and one modulo two becomes **0.9**.

9.5.5 Destructor for bit_vector

The **bit_vector** destructor needs no code body since there is no new data members. We will use the automatically generated default destructor which calls the **bit_array** class destructor to handle the memory management of **bit_vector**.

9.6 The bit_matrix Class

A thorough discussion on the design of matrix classes was presented in Chapter 5. It is convenient to model the **bit_matrix** class via a C array of **bit_vectors**, representing the rows of the matrix. The data member part of the **bit_matrix** class definition is:

```
class bit_matrix {
private:
    struct  mod_mat_rep  {
        bit_vector* m;
        int  rowdim;
        int  coldim;
        int  row_no_bytes;
        int   row_extra_bits;
        int  mrefs;
    }   *mp;
    ...
};
```

As usual, the matrix object is only an encapsulated pointer to a **rep** object that manages the deep memory. It is convenient to reuse all the bitwork that was previously done, and for this reason the matrix is represented by having the data member **m** point to the **0**th bit_vector in an array, created by **new bit_vector[rowdim]**, of **bit_vectors**. In other words, a **bit_matrix** is a C++ style array of the **bit_vectors**, making up the rows.

9.6.1 Bit-Component Access

Conveniently, the bracket **operator[]** can be used to return a reference to the appropriate row. If **A** is a **bit_matrix**, then **A[i]** returns a reference to the bit_vector representing the **i**th-row of **A**. Proxies are used to use A[i][j] with the A[i] bit vector.

9.6.2 The Class Definition for bit_matrix

Now is a good time to present the entire class definition for **bit_matrix**:

```
class bit_matrix  {
```

```
private:
    struct mod_mat_rep {
        bit_vector* m;
        int rowdim;
        int coldim;
        int row_no_bytes;
        int row_extra_bits;
        int mrefs;
    }  *mp;
public:
    int rowsize() const {return mp->rowdim;}
    int colsize() const {return mp->coldim;}
    bit_matrix(int , int) ;
    bit_matrix(int**, int, int);
    bit_matrix();
    bit_vector operator*(const bit_vector&) const;
    bit_matrix operator*(const bit_matrix&) const;
    bit_matrix operator*(unsigned char) const;
    bit_matrix operator+(const bit_matrix&) const;
    bit_matrix& operator=(const bit_matrix&);
    bit_matrix& operator*=(unsigned char);
    bit_matrix& operator+=(const bit_matrix&);
    bit_matrix(const bit_matrix&);
    bit_vector& operator[](int) const;
    bit_vector row(int) const;
    bit_vector column(int) const;
        unsigned char elem(int i, int j) const {
        if (mp->m[i].test_bit(j)) return one_mod_2; else return zero_mod_2;
    }
    bit_matrix deepcopy() const;
    int operator==(const bit_matrix&) const;
    int operator!=(const bit_matrix&) const;
    bit_matrix transpose() const;
    bit_matrix gauss_jordan() const;
    friend bit_matrix operator*(unsigned char, const bit_matrix&);
    friend ostream& operator<<(ostream&, const bit_matrix&);
    friend istream& operator>>(istream&, bit_matrix&);
    static bit_matrix zero(int, int);
    static void error(char *p);
};
```

9.6.3 Constructors for bit_matrix

The structure of the class **bit_matrix** is complicated enough to justify looking at the various constructors. First, consider the constructor **bit_matrix(int nrow, int ncol)**. First it creates a **mod_mat_rep**, using the global **new**. The address of this **rep** becomes the value of the only data member of **bit_matrix**. All the management information will be stored in the **mod_mat_rep**. The data members **rowdim**, **coldim**, **row_no_bytes**, and **row_extra_bits** of the **rep** are initial-

ized with the values determined by **nrow** and **ncol**. The components of the matrix will be stored in the array of **bit_vectors** that make up the rows. A pointer to this array of rows must be created. The statement **new bit_vector[nrow]** creates a C array of bit_vectors and returns a pointer that becomes the value of the data member **m** of the **rep**. The definition for this constructor is:

```
bit_matrix::bit_matrix(int nrow, int ncol) {
    mp = new mod_mat_rep;
    mp->rowdim = nrow;
    mp->coldim = ncol;
    mp->row_no_bytes = (ncol - 1)/8 + 1;
    mp->row_extra_bits = ncol%8;
    mp->m = new bit_vector[nrow]; // generates array of default vectors
    unsigned char* uptr;
    for (int i = 0; i < nrow; i++ )  {
        uptr = new unsigned char[mp->row_no_bytes];
        mp->m[i].assign(ncol, uptr);
    }
}
```

Now, we must discuss this **new** call. When **new** is called in the form **new CLASSNAME[n]**, memory is allocated on the heap to hold the data members of **n** contiguous objects of type **CLASSNAME**, and then the default constructor is called for each object in this dynamic array of **n** objects. What does all this mean? A default constructor for type **CLASSNAME** better have been written. In the case of types that represent C++ arrays, this default constructor cannot do too much because it cannot use the sizes of an array. It won't be able to allocate the deep memory that holds the components of the array; this is dependent on knowing the **dimension**. Thus, all a default vector constructor can do is allocate memory for a **bit_array_rep** and a pointer to that **bit_array_rep**. Later on, with a function that is not a constructor, we will have to create memory for vector components and then assign values into the data members of a **bit_array_rep**. Let's take a look at the **bit_vector** default constructor:

```
bit_vector::bit_vector()  :  bit_array()  {}
```

It merely calls the default constructor of **bit_array**. Maybe the code body of the default constructor **bit_array::bit_array()** will be more illuminating:

```
bit_array::bit_array() {
    rep = new bit_array_rep;
    rep->no_bytes = 0;
    rep->refs = 1;
    rep->dimension = 0;
    rep->extra_bits = 0;
    rep->ptr = 0;
}
```

As you can see, the **rep** is created and then initialized with null values. Whatever func-

tion calls this default constructor has to be able to go into and attach some deep memory to **rep->ptr** and assign valid values to the other **bit_array_rep** data members. The public member function **void assign(int, unsigned char*)** does this job, but does not create the deep memory.

This function **assign** should be used with extreme care; it was kept public in order that the **bit_matrix** member functions could access it. But also, any user defined function that wants to create an array of **bit_vectors** or **bit_arrays** will need it to assign usable values into the vector **reps**.

Recall the protection characteristics in the class definition. Since **assign** is in the public interface, if used improperly it could cause errors. As mentioned above, any function creating an array of vectors needs an **assign** to go with it; the programmer cannot be careless here. If the member functions of **bit_matrix** were the only functions that were ever going to create an array of **bit_vectors**, then we should make **assign** in **bit_array** protected and then in the class definition of **bit_vector** declare **bit_matrix** to be a friend of **bit_vector**. Prior to the class definition of **bit_vector** we must put the statement **class bit_matrix**; thus, the compiler knows that **bit_matrix** is a class. If **assign** were protected in **bit_array**, then the public inheritance would keep it **protected** in **bit_vector**. Then any friend of **bit_vector**, for example **bit_matrix**, could use the private and protected members of **bit_vector**. **assign** is public in **bit_array** and there is no access problem. All this shows that the class designer must think ahead about what functions might be needed to derive off or even use the classes. Of course, if you are given the complete source you can modify the original; but remember, anytime the original is touched, there is a good chance that the functional design might be corrupted. That is why derivation and composition are valuable. For proper reuse, there needs to be an available **assign**.

After that detour, let's jump back and finish up the basic **bit_matrix** constructor. It loops through these blank bit_vectors in the default array, for each vector (1) creating an array of deep memory and (2) then calling **bit_array::assign** to fill in the **bit_array_reps** with valid values.

It would be convenient to initialize a **bit_matrix** from a C two dimensional array of **0-1** integers such as **int a[7][4]**. Clearly this would make a nice constructor. The prototype for this constructor is: **bit_matrix::bit_matrix(int** ptr_ptr_int, int nrow, int ncol)**. This is a work around for the C two dimensional array problem. It would have been nice if two dimensional arrays in C were implemented with the name of the array representing an address where a contiguous array of pointers are stored. The **a[i]** would be a pointer to the zeroth integer in an array of integers representing the ith row. Then **a[i][j]** (the same as (**a[i])[j]**) would be dereferenced as the **j**th component of the **a[i]** vector. C does not interpret the double brackets in this pointer manner. **a** is simply the address of where **a[0][0]** is stored.

The problem boils down to: How can we pass in a C two dimensional array to a function that would allow for arrays of any size?. Remember C arrays are not like C++ vector objects; they don't encapsulate their size. The row and column sizes must be passed in separately. The next question is: How do we type the formal parameter without getting a compile time error?. Many compilers accept a double pointer as a formal parameter. So, we can now pass in the name of a C two dimensional array to a function. But this name is not a double pointer (it does not point to an array of int pointers). Inside of the code body for the constructor, we must type-cast it to an integer pointer. This is fine since it does point to the element **a[0][0]**. It should be noted that the compiler would not accept a formal parameter of integer pointer type. Now, a C two dimensional array is stored in row major order with all components contiguous. Therefore, if **ptr_int** is the integer pointer type cast of **ptr_ptr_int**, then **ptr_int[k+j*ncol]** refers to **a[j][k]**. This is the fix. The loop in the constructor that does the job is:

```
int* ptr_int = (int*)ptr_ptr_int;
for (int j = 0; j < nrow; j++)
        for (int k = 0; k < ncol; k++)
                (*this)[j][k] = (unsigned char)(ptr_int[k + j*ncol] );
    //
```

A copy-initializer for bit_matrix is provided; it is the standard **ref** counting copy-initializer.

9.6.4 Destructor for bit_matrix

The destructor for bit_matrix is worth talking about since an array of pointers to objects of a user defined class **(bit_vector**) is part of the data for bit_matrix. The code for the destructor is:

```
bit_matrix::bit_matrix()   {
    if   (--mp->mrefs == 0)   {
        delete[]  mp->m;
        delete  mp;
    }
}
```

As before if the **refs** count drops to zero, then the deep memory has to be destroyed. Now **delete[]** is being applied to a pointer to the zeroth object in an array of objects (in this case, **bit_vectors**). If those objects have a destructor (one that does something), then the destructor is called for each object in the array. In this case, each bit_vector has its destructor called. This essentially frees the component memory of each vector and the **rep** memory for each vector. Remember, in this book, the combination of component memory and **rep** memory of the vector is refereed to as the deep memory. After all the deep memory is freed for all the vectors, the memory containing the array of bit_vectors is freed. Realize the array of bit_vectors is an array of pointers to **bit_array_reps**. Actually, **delete[]** is doing a lot of work here. One does not put an argument in the **[]** when one calls **delete[]**; the executing program has a table of addresses and knows how much memory was allocated at that address. If it knows the type at that address (and thus the size of an object), it could calculate the number of objects in an array starting at that address. Individual compiler manuals don't usually spell these details out. The bottom line is that the heap table provides the information on **[]**.

Now, that all of the bit_vectors have been destroyed, we can free up the memory of the **mod_mat_rep**. The **mod_mat_rep** was not provided with a destructor; thus, **delete mp** frees the memory up that **mp** points to. No destructor is called for **mp** since the data members of **mod_mat_rep** are not user defined types (thus, no destructor was automatically created for **mod_mat_rep**).

These destructors work nicely. Realize that when the array of **bit_vectors** was initially created, they had null pointers to the deep memory that held their components. Later on, we assign memory to these pointers and the storage requirements change. What is nice about the destructors is that they are able to destroy vectors that may have different storage configurations

from that of their initial creations. The pointer and heap table work together to make it possible.

9.7 C++ Files

9.7.1 Bit Arrays

9.7.1.1 File *bitarray.h*

```
// file bitarray.h
// class definitions for bit_array
// April 27, 2003 - Proxy Funcionality for operatot[]
// April 10, 2003
#ifndef BIT_ARRAY_H
#define BIT_ARRAY_H
// #include <iostream>
#include <iosfwd>
using namespace std;

static unsigned char zero_mod_2 = 0;
static unsigned char one_mod_2 = 1;
static unsigned char bit_check[] = {1,2,4,8,16,32,64,128};
static unsigned char bit_mask[] = {255,1,3,7,15,31,63,127};
static unsigned char front_end_mask[] = {255,254,252,248,240,224,192,128};

class bit_array {
private:
struct bit_array_rep {
        int dimension;
   int no_bytes;
   int extra_bits;
   int refs;
   unsigned char* ptr;
}* rep;
protected:
   unsigned char* address_array() const { return rep->ptr; }
   struct bit_array_rep* address_array_rep() const { return rep; }

public:
        class BitProxy {
        public:
                BitProxy(bit_array& ba, int position);
```

```
            BitProxy& operator=(const BitProxy& rhs);
            BitProxy& operator=(unsigned char m2);
            operator unsigned char() const;
        private:
            bit_array& the_bit_array;
            int bit_position;
        };

public:
  void assign(int , unsigned char*);
  int size() const { return rep->dimension; }
  int byte_size() const { return rep->no_bytes; }
  bit_array(int); // constructor
  bit_array();
  bit_array(const bit_array&);  // copy-initializer
  bit_array& operator=(const bit_array&);  // assignment equals
  ~bit_array() {
    if (--rep->refs == 0) {
      delete[] rep->ptr;
      delete rep;
    }
  }  //destructor
  bit_array operator&(const bit_array&) const; // and operator
  bit_array operator|(const bit_array&) const; // or operator
  bit_array operator^(const bit_array&) const; // exclusive or operator
  bit_array operator~() const; // complement operator
  int operator==(const bit_array&) const; // logical equals
  int operator!=(const bit_array&) const; // logical not equals
  int test_bit(int n) const { return (rep->ptr[n/8] & bit_check[n%8]);}
  void set_bit(int n) const {
        rep->ptr[n/8] = rep->ptr[n/8] | bit_check[n%8];}
  void clear_bit(int n) const {
    rep->ptr[n/8] = rep->ptr[n/8] & ~(bit_check[n%8]);}
  static void error(char*);
  unsigned char elem(int i) const {
        if (test_bit(i) ) return one_mod_2; else return zero_mod_2;
  }
  static int bit_count(unsigned char uch) ;
  int weight( ) const;
  bit_array deepcopy() const;
  // proxy functionality
  const BitProxy operator[](int position) const;
```

```
  BitProxy operator[] (int position);
  friend class BitProxy;
  friend ostream& operator<<(ostream&, const bit_array&);
  friend istream& operator>>(istream&, bit_array&);
};
#endif BIT_ARRAY_H
```

9.7.1.2 File *bitarray.cpp*

```
// file bitarray.cpp
// April 10,2003
// implementations for bit_array
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream>
// #include "tracer.h"
#include "bitarray.h"

bit_array::bit_array(int size) {
  rep = new bit_array_rep;
  rep->no_bytes = (size-1)/8 + 1;
  rep->refs = 1;
  rep->dimension = size;
  rep->extra_bits = size%8;
  rep->ptr = new unsigned char[rep->no_bytes];
  for (int i = 0; i < rep->no_bytes ; i++) rep->ptr[i] = '\0';
}

bit_array::bit_array() {
  rep = new bit_array_rep;
  rep->no_bytes = 0;
  rep->refs = 1;
  rep->dimension = 0;
  rep->extra_bits = 0;
  rep->ptr = 0;
}

void bit_array::assign(int dim, unsigned char* uptr) {
  rep->no_bytes = (dim -1)/8 + 1;
  rep->refs = 1;
```

```
  rep->dimension = dim;      // correction
  rep->extra_bits = dim%8;
  rep->ptr = uptr;
}

bit_array::bit_array(const bit_array& ba) {
  rep = ba.rep;
  rep->refs++;
}

bit_array bit_array::deepcopy() const {
//   Tracer tr("bit_array::deepcopy");
  bit_array result(size());
  // for (int i = 0; i < size(); i++) result.lvalue(i, elem(i) );
  for (int i = 0; i < size(); i++) result[i] =  elem(i);
  return result;
}

bit_array& bit_array::operator=(const bit_array& ba) {
//    Tracer tr("bit_array::op=(bit_array)");
  if ( rep->ptr != ba.rep->ptr) { // if assignment to self don't copy
    if (--rep->refs == 0) {
          delete[] rep->ptr;
          delete rep;
    }   // correction
    rep = ba.rep;
    rep->refs++;
  }
  return *this;
}

bit_array bit_array::operator&(const bit_array& ba) const {
  bit_array result(rep->dimension);
  for (int i = 0; i < rep->no_bytes; i++) result.rep->ptr[i] = rep->ptr[i] &
    ba.rep->ptr[i];
  return result;
}

bit_array bit_array::operator|(const bit_array& ba)  const {
  bit_array result(rep->dimension);
  for (int i = 0; i < rep->no_bytes; i++)
    result.rep->ptr[i] = rep->ptr[i] | ba.rep->ptr[i];
```

```
  return result;
}

bit_array bit_array::operator^(const bit_array& ba) const {
  bit_array result(rep->dimension);
  for (int i = 0; i < rep->no_bytes; i++)
    result.rep->ptr[i] = rep->ptr[i] ^ ba.rep->ptr[i];
  return result;
}

bit_array bit_array::operator~()  const                    {
  bit_array result(rep->dimension);
  for (int i = 0; i < rep->no_bytes; i++)
    result.rep->ptr[i] =  ~(rep->ptr[i]);
  return result;
}

int bit_array::operator==(const bit_array &ba) const {
//    Tracer tr("bit_array::op==(bit_array)");
  if (ba.rep->dimension != rep->dimension) return 0;
  for (int i = 0; i < rep->no_bytes-1; i++)
    if (ba.rep->ptr[i] != rep->ptr[i]) return 0;
   // check the relevant bits in high byte
  if ( (bit_mask[rep->extra_bits] & ba.rep->ptr[rep->no_bytes-1]) !=
    ( bit_mask[rep->extra_bits] & rep->ptr[rep->no_bytes-1]) ) return 0;
  return 1;
}

int bit_array::operator!=(const bit_array &ba)  const {
//    Tracer tr("bit_array::op!=");
  if (*this == ba) return 0; else return 1;
}

void bit_array::error(char* p) {
  cerr << p << "\n";
  exit(1);
}

int bit_array::weight() const {
//   Tracer("bit_array::weight");
  int result = 0;
  for (int i =0; i < rep->no_bytes - 1; i++) {
```

```
    unsigned char uch = address_array()[i];
    result += bit_count(uch);
  }
  result += bit_count( (address_array()[rep->no_bytes-1] ) &
    bit_mask[rep->extra_bits] );
  return result;
}

int bit_array::bit_count(unsigned char uch)  {
  static int count[256];
  static char init = 1;
  int temp_count;
  if (init) {
    for (int j = 0; j <256; j++) {
      temp_count = 0;
          unsigned char jj = (unsigned char)j;
          for (int k = 0; k < 8; k++)
              if (jj  & bit_check[k] ) temp_count++;
          count[j] = temp_count;
    }
    init = 0;
  }
  return count[uch];
}

ostream& operator<<(ostream& s, const bit_array& ba) {
  s << "\n";
  for (int i = 0; i < ba.size(); i++) {
     // if (ba[i] == zero_mod_2) s << " 0" ; else s << " 1";
     if (0 == ba[i]) s << " 0" ; else s << " 1";
  }
  s << "\n";
  return s;
}

istream& operator>>(istream& s, bit_array& ba) {
  int buffer;
  unsigned char uch;
  for (int i = 0;i < ba.size(); i++) {
    s >> buffer;
    uch = *( (unsigned char*)&buffer );  // get low-order byte
    // ba.lvalue(i, uch & one_mod_2);
```

```
    ba[i] = uch & one_mod_2;
  }
  return s;
}


const bit_array::BitProxy bit_array::operator[](int position) const  {
   return BitProxy(const_cast<bit_array&>(*this), position);
}

bit_array::BitProxy bit_array::operator[] (int position) {
        return BitProxy(*this, position);
}


bit_array::BitProxy::operator unsigned char() const {
  // check the bit_array in the proxied position
        if (the_bit_array.test_bit(bit_position) ) return 1;
        else return 0;
}

bit_array::BitProxy::BitProxy(bit_array& ba, int position) :
the_bit_array(ba), bit_position(position) {}



bit_array::BitProxy& bit_array::BitProxy::operator=(const BitProxy& rhs) {

                // go into the bit_array and change the bit
                // convert right hand side to an integer
                unsigned char temp_int = (unsigned char)rhs;
                if (temp_int) the_bit_array.set_bit(bit_position);
                else the_bit_array.clear_bit(bit_position);
                return *this;
}

bit_array::BitProxy& bit_array::BitProxy::operator=(unsigned char m2)  {
        if (m2 & one_mod_2) the_bit_array.set_bit(bit_position);
                else the_bit_array.clear_bit(bit_position);
                return *this;
}
```

9.7.2 Bit Sets

9.7.2.1 File *bitsets.h*

```
// file bitsets.h

#ifndef BITSETS_H
#define BITSETS_H
#include "bitarray.h"

class ASCII_set : public bit_array {
public:
   ASCII_set() : bit_array(128) {}
   ASCII_set(char* s);
   ASCII_set(const ASCII_set& bs) : bit_array(bs) { } //copy-initializer
   ASCII_set(const bit_array& ba) : bit_array(ba) {
        if (ba.size() != 128)
           error("\nIn ASCII_set(ba) ba does not have dimension 128\n");
   }
   ASCII_set& operator=(const ASCII_set& bs){
        bit_array::operator=(bs);
        return (*this);
   }
   ASCII_set& operator=(const bit_array& ba){
        if (ba.size() != 128)
              error("\nIn ASCII_set(ba) ba does not have dimension 128\n");
        bit_array::operator=(ba);
        return (*this);
     }
   ~ASCII_set() { }
   int test_char(unsigned char ch) { return test_bit(ch); }
   void remove_char(unsigned char ch) { clear_bit(ch); }
   void insert_char(unsigned char ch) { set_bit(ch); }
   ASCII_set operator+(ASCII_set& bs) { return *this | bs; } // set union
     ASCII_set operator-(ASCII_set& bs) { return *this ^ (*this * bs); }
     ASCII_set operator*(ASCII_set& bs) { return *this & bs;} // set intersection
   char filter_char();
   ASCII_set set_minus_string(char*);
   ASCII_set set_plus_string(char*);
   static int in_string(char,char*);
   friend ostream& operator<<(ostream& s,ASCII_set& bs);
   friend istream& operator>>(istream& s,ASCII_set& bs);
};

class bit_set : public bit_array{
public:
   bit_set(unsigned int n) : bit_array(n) { }
```

```
  bit_set(const bit_set& bs) : bit_array(bs) { } //copy-initializer
  bit_set(const bit_array& ba) : bit_array(ba){ }
  bit_set& operator=(const bit_set& bs) {
       bit_array::operator=(bs);
       return (*this);
  }
  bit_set& operator=(const bit_array& ba) {
     bit_array::operator=(ba);
       return (*this);
  }
  ~bit_set() { }
  bit_set operator+(bit_set& bs) { return *this | bs; } // set union
  bit_set operator-(bit_set& bs) { return *this ^ (*this * bs); }
  bit_set operator*(bit_set& bs) { return *this & bs;} // set intersection
};

#endif BITSETS_H
```

9.7.2.2 File *bitsets.cpp*

```
// implementation file - bitsets.cpp

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include "tracer.h"
#include "bitsets.h"

ASCII_set::ASCII_set(char* s) : bit_array(128) {
  char ch;
  while (ch = *s++) insert_char(ch);
}

char ASCII_set::filter_char() {
  char ch;
  while (test_char(ch = getch()) == 0)   putchar('\7'); /* if not OK
                                                 sound bell */
  if ( (ch>=' ') && (ch<='~') ) putchar(ch);    /* display it */
  return ch;
}

ASCII_set ASCII_set::set_plus_string(char* string) {
  unsigned char ch;
  ASCII_set result(this->deepcopy());
  for (int i = 0; ch = *(string+i); i++)  result.insert_char(ch);
  return result;
}
```

```
ASCII_set ASCII_set::set_minus_string(char* string) {
  unsigned char ch;
  ASCII_set result(this->deepcopy());
  for (int i = 0; ch = *(string+i); i++)  result.remove_char(ch);
  return result;
}

int ASCII_set::in_string(char ch, char* string) {
  char c;
  for (int i = 0; c = *(string+i); i++) if (ch == c) return 1;
  return 0;
}
```

9.7.2.3 File *intread.cpp*

```
/* keyboard filtering in "C++" for DOS */
/* modifications done March 29, 1994 */

#include "bitsets.h"
#include "string.h"
#include <stdio.h>

#define SETSIZE 16

intread() {
    int count,flag1,j,max,number,posflag,signflag;
  char *w,*ptr,ch,
  instring[10];
  int maxlen = 6;
  ASCII_set DIGITS("0123456789");
  ASCII_set SIGNS("+-");
  ASCII_set BSRET("\10\15");
  ASCII_set BSDIGITS = DIGITS.set_plus_string("\b");
  ASCII_set BSPRETDIGITS = BSRET + DIGITS;
  ASCII_set DIGITS_AND_SIGNS = DIGITS + SIGNS;
  w = instring; /* initialize to instring */
  // flag1 = 0 means the string is acceptable for a 2-byte signed integer
  do {
    flag1 = 1;
    count = 0 ; /*  no chars entered yet */
    do {
      if (count == 0) {
      // if a "+" sign or no sign is entered conditions are set as below
        max = maxlen - 1;
        signflag = 0;
        posflag = 1;
        ch = DIGITS_AND_SIGNS.filter_char();
```

```
            if (SIGNS.test_char(ch)) {  /* tests for "+","-" */
               if (ch == '-') posflag = 0; /* negative entry */
               max = max + 1;       /* distinguishes from + */
               signflag = 1;  /*  sign flag == 1 means + or - entered */
            }
            count++;          /* acceptable character count */
         }
         else {
                if (count == max) { /* allow bkspces/cr's in okset */
            ch = BSRET.filter_char();
            count++;
            } else
               if ( (count == 1) && (signflag) ) {
                  ch = BSDIGITS.filter_char();
                  count++;
               } else {
                  ch = BSPRETDIGITS.filter_char();
                  count++;
               }
         }
         if (DIGITS_AND_SIGNS.test_char(ch)) *w++ = ch; /* put in array */
         else if (ch == '\10') {     /* if backspace */
            printf("\b%c\b",' ');               /* clean up */
            count -= 2;
            w -= 1;
         }
      } while (ch != '\15');               /* so long as no CR */
      *w++ = '\0';                        /* add '\0' at end  */
      w -= count; /* address to string that contains the  number  */
      if (signflag) ptr = w + 1; else ptr = w; /* - had one more entry */
      if ( ( (signflag) && (count < 7) ) || ( (signflag == 0) &&
         (count < 6 ) ) ) flag1 = 0;
      else  /* tests if within -32768 <--> +32767 limits */
         flag1 =  0 < strcmp(ptr, posflag ? "32767" : "32768");
      if (flag1) for (j=1;j < count ;j++) printf("\b%c\b\7",' ');
   } while (flag1); /* change made on 6-5-87 */
   number = 0;
       /* change made on 6-5-87 */
     while  (ch = *ptr++)  number = number*10 + ch - '0';
   if (posflag == 0) number = -number;
   return(number);
}

int main() { /* used for test purposes */
   int value;
   char ch;
   char *prompt,*question;
   int i;

/* the following tests some of the set functions */
```

```
   ASCII_set tset1("+-0123456789");
   ASCII_set tset2("+-");
   ASCII_set tset3("?!");
   ASCII_set tset4 = tset2*tset1;
   ASCII_set tset5 = tset3+tset2;
   ASCII_set tset6 = tset2-tset1;
   printf("\n");

    /* display results */

   for (i=0;i<128;i++) if (tset4.test_char(i)) putchar(i);
   printf("\n");
   for (i=0;i<128;i++) if (tset5.test_char(i)) putchar(i);
   printf("\n");
   for (i=0;i<128;i++) if (tset6.test_char(i)) putchar(i);

/* this completes the test of the above */

/* the following tests intread() */
   ASCII_set yesno("yYnN");
   ASCII_set yes("yY");
   prompt = "please enter an integer ";
   question = "do you want to do it again? y/n ";
   do {
      printf("\n%s",prompt);
      value = intread();
      printf("\n%d",value);
      printf("\n%s",question);
      ch = yesno.filter_char();
   } while (yes.test_char(ch));
   printf("\n");
   return 1;
/* end of intread() test */
}
```

9.7.3 Bit Vectors and Matrices

9.7.3.1 File *bitref.h*

```
// April 27, 2003
// class definitions for bit vectors and matrices
#ifndef BIT_CLASSES_WITH_REFERENCES
#define BIT_CLASSES_WITH_REFERENCES
// #include <iostream>
```

```
#include <iosfwd>
#include "bitarray.h"

class bit_vector : public bit_array  {
public:
        bit_vector() : bit_array() {}
        bit_vector(int n) : bit_array(n) {}
        bit_vector(int *, int);
        bit_vector(const bit_vector& mv) : bit_array(mv) { }
        bit_vector(const bit_array& ba) : bit_array(ba) {}
        bit_vector& operator=(const bit_vector& mv) {
                bit_array::operator=(mv);
                return (*this);
        }
        bit_vector& operator=(const bit_array& ba) {
                bit_array::operator=(ba);
                return (*this);
        }
        ~bit_vector() { }
        bit_vector deepcopy() const;
        static bit_vector zero(int);
        bit_vector operator+(const bit_vector& mv) const { return *this ^ mv; }
        unsigned char operator*(const bit_vector& mv) const;
        bit_vector operator*(unsigned char) const;
        bit_vector& operator+=(const bit_vector&);
        bit_vector& operator*=(unsigned char);
        int hamming_distance(const bit_vector& mv) const { return (*this + mv).weight();}
        float* mod2_to_float() const;
        friend bit_vector operator*(unsigned char,const bit_vector&);
        friend ostream& operator<<(ostream&,const bit_vector&);
};

class bit_matrix {
private:
struct mod_mat_rep {
        bit_vector* m;
        int rowdim;
        int coldim;
        int row_no_bytes;
        int row_extra_bits;
        int mrefs;
}  *mp;
```

```
public:
        int rowsize() const {return mp->rowdim;}
        int colsize() const {return mp->coldim;}
        bit_matrix(int , int) ;
        bit_matrix(int**, int, int);
        ~bit_matrix();
        bit_vector operator*(const bit_vector&) const;
        bit_matrix operator*(const bit_matrix&) const;
        bit_matrix operator*(unsigned char) const;
        bit_matrix operator+(const bit_matrix&) const;
        bit_matrix& operator=(const bit_matrix&);
        bit_matrix& operator*=(unsigned char);
        bit_matrix& operator+=(const bit_matrix&);
        bit_matrix(const bit_matrix&);
        bit_vector& operator[](int) const;
        bit_vector row(int) const;
        bit_vector column(int) const;
        unsigned char elem(int i, int j) const {
                if (mp->m[i].test_bit(j)) return one_mod_2; else return zero_mod_2;
        }
        bit_matrix deepcopy() const;
        int operator==(const bit_matrix&) const;
        int operator!=(const bit_matrix&) const;
        bit_matrix transpose() const;
        bit_matrix gauss_jordan() const;
        friend bit_matrix operator*(unsigned char, const bit_matrix&);
        friend ostream& operator<<(ostream&, const bit_matrix&);
        friend istream& operator>>(istream&, bit_matrix&);
        static bit_matrix zero(int, int);
        static void error(char *p);
};
#endif BIT_CLASSES_WITH_REFERENCES
```

9.7.3.2 File *bitref.cpp*

```
// implementation file - bitref.cpp
// April 10,2003
// April 27, 2003
#include <stdlib.h>
#include <iostream>
#include "tracer.h"
#include "bitref.h"
```

```
bit_vector::bit_vector(int* int_ptr, int dim) : bit_array(dim) {
   for (int i = 0; i < dim; i++) (*this)[i] = (unsigned char)(int_ptr[i]);
}

unsigned char bit_vector::operator*(const bit_vector& mv) const {
        //    Tracer tr("mod_vec::op*(mod_vec)");
        bit_vector temp(*this & mv);
        int value= temp.weight();
        return (value%2);
}

bit_vector bit_vector::operator*(unsigned char uch) const {
        //   Tracer tr("mod_vec::op*(char)");
        if (bit_check[0] & uch) return deepcopy();
        else return zero(size());
}

bit_vector& bit_vector::operator+=(const bit_vector& mv) {
        Tracer tr("mod_vec::op+=(mod_vec)");
        if (size() != mv.size() ) error("vectors have diff. dim in mod_vec+= ");
   for (int i = 0; i < size(); i++) (*this)[i] = elem(i) ^ mv[i] ;
        return *this;
}

bit_vector& bit_vector::operator*=(unsigned char uch) {
  Tracer tr("mod_vec::op*=(char)");
  if (bit_check[0] & uch) return *this;
  else {
                for (int i = 0; i < size(); i++) (*this)[i] = zero_mod_2;
                return *this;
  }
}

bit_vector bit_vector::zero(int n) {
//   Tracer tr("mod_vec::zero");
  bit_vector result(n);
  for (int i = 0; i < result.byte_size(); i++)
     result.address_array()[i] = '\0';
  return result;
}
```

```
bit_vector bit_vector::deepcopy() const  {
//   Tracer tr("mod_vec::deepcopy");
  bit_vector mv((*this).bit_array::deepcopy() );
  return mv;
}

bit_vector operator*(unsigned char uch, const bit_vector& mv) {
  return mv*uch;
}

ostream& operator<<(ostream& s, const bit_vector& ba) {
        s << "\n";
        for (int i = 0; i < ba.size(); i++) {
                if (ba[i] == zero_mod_2) s << " 0" ; else s << " 1";
        }
        s << "\n";
        return s;
}

bit_matrix::bit_matrix(int nrow, int ncol) {
        mp = new mod_mat_rep;
        mp->rowdim = nrow;
        mp->coldim = ncol;
        mp->row_no_bytes = (ncol-1)/8 + 1;
        mp->row_extra_bits = ncol%8;
        mp->m = new bit_vector[nrow];  // generates array of default vectors
        unsigned char* uptr;
        for (int i = 0; i < nrow; i++ ) {
                uptr = new unsigned char[mp->row_no_bytes];
                mp->m[i].assign(ncol,uptr);
        }
}

bit_matrix::bit_matrix(int** ptr_ptr_int, int nrow, int ncol) {
        mp = new mod_mat_rep;
        mp->rowdim = nrow;
        mp->coldim = ncol;
        mp->row_no_bytes = (ncol-1)/8 + 1;
        mp->row_extra_bits = ncol%8;
        mp->m = new bit_vector[nrow];  // generates array of default vectors
        unsigned char* uptr;
        for (int i = 0; i < nrow; i++ )  {
```

```
                uptr = new unsigned char[mp->row_no_bytes];
                mp->m[i].assign(ncol,uptr);
        }
        int* ptr_int = (int*)ptr_ptr_int;
        for (int j = 0; j < nrow; j++)
                for (int k = 0; k < ncol; k++)
                        (*this)[j][k] = (unsigned char)(ptr_int[k + j*ncol] );
}

bit_matrix::~bit_matrix() {
        //      Tracer tr("~mod_mat");
        if (--mp->mrefs == 0) {
                delete[] mp->m; //calls destructor for each "row_vector"
                delete mp;
        }
}

bit_vector bit_matrix::row(int h) const {
        Tracer tr("mod_mat::row");
        bit_vector result(colsize() );
        for (int i = 0; i < colsize(); i++) {
                unsigned char uch = ((mp->m)[h])[i];
                result[i] = uch;
        }
        return result;
}

float* bit_vector::mod2_to_float()              const {
        float* dptr = new float[size()];
        for (int i = 0; i < size(); i++) {
                if ( (*this)[i] == one_mod_2) dptr[i] = 0.9;
                else dptr[i] = 0.1;
        }
        return dptr;
}

bit_vector bit_matrix::operator*(const bit_vector& mv) const {
        //   Tracer tr("mod_mat::op*(mod_vec)");
  bit_vector result(rowsize());
        for (int i = 0; i< rowsize(); i++) {
                unsigned char uch = mv*((*this)[i] );
      result[i] = uch;
```

```
  }
  return result;
}

bit_matrix bit_matrix::operator+(const bit_matrix& mm) const {
        //   Tracer tr("mod_mat::op+(mod_mat)"):
  if ( (rowsize() != mm.rowsize()) || (colsize() != mm.colsize() ) )
    error("matrices of different dimensions in op+");
  bit_matrix result(rowsize(), colsize());

   for (int i = 0; i < rowsize() ; i++)    result[i] = (*this)[i] + mm[i];
  return result;
}

bit_matrix bit_matrix::operator*(unsigned char uch) const {
//   Tracer tr("mod_mat::op*(char)");
        bit_matrix result(rowsize(), colsize());
        for (int i = 0; i < rowsize() ; i++) result[i] = ((*this)[i]) * uch;
        return result;
}

bit_matrix bit_matrix::operator*(const bit_matrix& mm) const {
//   Tracer tr("mod_mat::op*(mod_mat)");
        if  (colsize() != mm.rowsize() )
                error("matrices of incompatible dimensions in op*");
        bit_matrix result(rowsize(), mm.colsize() );
        for (int i = 0; i < rowsize(); i++)
                for (int j = 0; j < mm.colsize(); j++)
                        result[i][j] =  (*this)[i] * mm.column(j) ;
        return result;
}

bit_matrix& bit_matrix::operator+=(const bit_matrix& mm) {
//   Tracer tr("mod_mat::op+=(mod_mat)");
        if ( (rowsize() != mm.rowsize()) || (colsize() != mm.colsize() ) )
                error("matrices of different dimensions in op+=");
        for (int i = 0; i < rowsize(); i++)
                for (int j = 0; j < colsize(); j++) (*this)[i][j] =  elem(i,j) ^ mm.elem(i,j) ;
        return *this;
}

bit_matrix& bit_matrix::operator*=(unsigned char uch) {
```

```
//    Tracer tr("mod_mat::op*=(char)");
        if (uch & one_mod_2) return *this;
        else {
                for (int i = 0; i < rowsize(); i++)
                        for (int j = 0; j < colsize(); j++)  (*this)[i][j] = zero_mod_2;
                return *this;
        }
}

bit_matrix::bit_matrix(const bit_matrix& mm) {
        mm.mp->mrefs++;
        mp = mm.mp;
}

bit_matrix& bit_matrix::operator=(const bit_matrix& mm) {
        //      Tracer tr("mod_mat::op=(mod_mat)");
        if (this->mp->m == mm.mp->m ) return *this;
        if (--mp->mrefs == 0) {
                delete[] mp->m; //calls destructor for each "row_vector"
                delete mp;
        }
        mm.mp->mrefs++;
        mp = mm.mp;
        return *this;
}

int bit_matrix::operator==(const bit_matrix& mm) const {
        //   Tracer tr("mod_mat::op==");
        if (rowsize() != mm.rowsize() ) return 0;
  if (colsize() != mm.colsize() ) return 0;
        for (int i = 0; i < rowsize() ; i++)   if ( (*this)[i] != mm[i]) return 0;
  return 1;
}

int bit_matrix::operator!=(const bit_matrix& mm) const {
        //   Tracer tr("mod_mat::op!=");
        if (*this == mm) return 0; else return 1;
}

bit_vector& bit_matrix::operator[](int i) const {
        return mp->m[i];
}
```

```
bit_vector bit_matrix::column(int j) const {
        //   Tracer tr("mod_mat::column");
        bit_vector result(rowsize());
   for (int k = 0; k < rowsize(); k++) result[k] = elem(k,j);
        return result;
}


bit_matrix bit_matrix::deepcopy() const {
//   Tracer tr("mod_mat::deepcopy");
        bit_matrix result(rowsize(),colsize() );
        for (int i = 0; i < rowsize(); i++)
                for (int j = 0; j < colsize(); j++) result[i][j] =  elem(i,j) ;
        return result;
}

bit_matrix bit_matrix::transpose() const {
//      Tracer tr("mod_mat::transpose");
        bit_matrix result(colsize(), rowsize());
        for (int i = 0; i < rowsize(); i++);
   for (int j = 0; j < colsize(); j++) result[i][j] =  elem(i,j) ;
        return result;
}

bit_matrix bit_matrix::gauss_jordan() const {
//         Tracer tr("gauss-jordan ");
        bit_matrix rm(deepcopy() );
        int numrows = rm.rowsize();
        int numcols = rm.colsize();
        unsigned char numzero = zero_mod_2;
        bit_vector rv(numcols);
        bit_vector w(numcols);
        int* ptrrc = new int[numrows];
        bit_vector cv(numrows);
        cout << rm;
        for (int i = 0; i < numrows; i++) {
                int j = 0;
                while ( ( j < numcols) && ( rm.elem(i,j) == numzero) ) j++;
                if ( j >= numcols) ptrrc[i] = -1;
                else {
                        ptrrc[i] = j;
```

```
                        cout << rm;
                        cv =  rm.column(j);
                        for ( int k = 0; k < numrows; k++)
                                if ( ( k != i) && ( cv[k] != numzero)) {
                  //            Tracer tr1("+= loop");
                                        rm[k] = rm[k] + rm[i];
                                        cout << rm;
                                }
                }
        }
        delete[] ptrrc;
        return rm;
}

bit_matrix operator*(unsigned char uch, const bit_matrix& mm) {
        Tracer tr("op*(char,mod_mat)");
  bit_matrix result(mm*uch);
  return result;
}

ostream& operator<<(ostream& s,const bit_matrix& mm) {
        s << "\n";
        for (int i = 0; i < mm.rowsize(); i++) {
    s << "\n";
    for (int j = 0; j < mm.colsize(); j++)
                if( mm[i][j] == zero_mod_2) s << " 0"; else s << " 1";
                  //  s << mm[i];
        }
        s << "\n";
        return s;
}

istream& operator>>(istream& s,bit_matrix& mm) {
  int buf;
  unsigned char uch;
        for (int i = 0; i < mm.rowsize(); i++) {
    cout << "\n enter row # " << i << "\n";
                for (int j = 0; j < mm.colsize(); j++) {
                        s >> buf;
                        uch = *((unsigned char*)&buf);
        mm[i][j] = uch & one_mod_2 ;
                }
```

```
        }
  return s;
}

bit_matrix bit_matrix::zero(int m, int n) {
        Tracer tr("mod_mat::zero");
  bit_matrix result(m,n);
  for (int i = 0; i < m; i++) result[i] = bit_vector::zero(n);
  return result;
}

void bit_matrix::error(char *p) {
  cerr << "\n" << p;
  exit(1);
}
```

EXERCISES

1. Determine what unsigned integer type uses 4 bytes of storage. Then implement **bit_array** and all its derived classes using this type in place of **unsigned char**.

2. Implement **bit_array** and all its derived classes using **unsigned int**, but use the **sizeof** function to transparently take into account whether the implementation uses 2 or 4 bytes for an **unsigned int**.