EECS 211 – Spring Quarter, 2010
Program 3

Due Wednesday, April 28th, 2010 at 11:59PM

In this first program in our project we will begin to implement two of the classes we need to model computers with disk drives that store files. The first class will model the disk drive itself; the second class will model a computer that contains a disk drive. In later assignments we will implement classes that represent different types of computers, such as laptops, file servers and print servers.

Background:

Modern mass storage devices such as disk drives are a part of every general purpose computer. In this assignment we will develop a C++ class that simulates these mass storage devices. This section provides some background and insight into these devices that may help you understand the assignment and the requirements.

The data storage area on disk drives is divided into basic blocks based on the physical nature of the drive. For example, the typical drive in a PC has the storage on circular "platters". The outer rim of the platter usually has timing marks so that as the platter spins the hardware can determine which portion of the surface is under the read/write heads. This leads to a partitioning of the surface into pie-shaped sections, each pie-shaped section corresponding to a fixed number of timing marks. The surface is further partitioned into a set of concentric circles, called tracks. This dual partitioning produces a set of arc-shaped areas called blocks, each block capable of storing a fixed number of bytes. (A typical block size is 256 bytes. In our project we will have blocks of size 64 bytes.) The set of blocks is numbered, starting from 0. If the block size is B and the number of blocks is N, the total number of bytes that can be stored is B*N.

The first blocks on a disk are grouped together into the so-called root directory. This directory contains information about the other files and directories on that disk. If the disk has been formatted as a system disk, the root directory also contains information and files your computer needs when it first starts up. We will not be concerned in this project about system information in a root directory, but later assignments will implement a primitive kind of root-level file system.

When a file is written to a disk, information about the file (for example, the name, number of bytes, where the actual data is stored, etc.) is stored in one place on the disk. The actual data is stored in another area. The file information likely does not require more than one block, and in fact a single block may contain the file information for several files. However, the actual data itself may be (typically is) many times larger than can fit in a single block. Therefore, the data is stored in a series of blocks. In modern operating systems these blocks need not be contiguous; the operating system, working in conjunction with the hardware, may use elaborate schemes to optimize performance of

data storage and retrieval and to prevent so-called disk fragmentation. In our project we will place the data in contiguous blocks for simplicity.

When a block is being used for a file, we say that block is allocated. When a file is deleted, the blocks that were allocated for that file become free and can be used later for another file. The hardware and/or operating system has to maintain some kind of data structure that tells at any given time which blocks are currently allocated and which are free. We will use a very simple structure, the bit map. Suppose a disk had 20 blocks. Recording the allocated/free status would require 20 bits or 3 bytes; in this case the last four bits of the third byte would not be used. If block j was allocated, then the j^{th} bit in the string of 20 bits would be set to 1; otherwise that bit would be set to 0.

Finally, because of the physical nature of the spinning of the surfaces and the read/write heads, data is transferred to or from the magnetic surface in whole blocks. Thus, the storing or retrieving of a file is actually accomplished as a series of block transfers to or from the surface of the disk. Note that storage and retrieval are distinct from allocating and freeing. The blocks for a file are allocated once, namely when the file is first created. Data within the file (that is, within some block being used by the file) may be retrieved or even changed many times, and of course the blocks remain allocated. The blocks are freed only if the file is deleted, and of course this also happens at most once per file.

Assignment:

(1) Your project for the assignment will consist of five files – disk_drive.h, disk_drive.cpp, machines.h, machines.cpp, and main.cpp.

(2) Declare and implement the following class. The declaration should be made in the file disk_drive.h, and the implementation done in the file disk_drive.cpp.

Class **diskDrive**:

- data members:
  - **int** – the number of 64-byte blocks of storage contained in the object.
  - **char** pointer – points to a set of bytes that represent a bit-map that will be used to tell which blocks are allocated and which are free.
  - **char** pointer – points to a block of storage of size 64*(number of blocks).
- function members:
  - A constructor which takes as argument an integer telling how many bytes of storage the disk should hold. This need not be a multiple of 64. This function computes and stores the number of blocks the disk will have. It then allocates enough bytes to hold that many blocks and the corresponding bit map. All the bytes in the bit map should be set to 0.
  - A destructor. This function frees the bit map and storage memory.

- A function called **print**. This function prints the number of blocks on the drive, the number of bytes in the bit map, and the actual bit map itself. The bit map should be printed in hexadecimal format.
- A function called **allocateBlock** which takes an integer representing a block number as argument and allocates (i.e., sets the corresponding bit in the bit map) the indicated block. If the argument does not represent a valid block number for that disk, print an error message.
- A function called **freeBlock** which takes an integer representing a block number as argument and frees (i.e., clears the corresponding bit in the bit map) the indicated block. If the argument does not represent a valid block number for that disk, print an error message.
- A function called **isBlockFree** which takes an integer representing a block number as argument and returns 1 if the indicated block is free and 0 if it is not. If the argument does not represent a valid block number for that disk, print an error message and return 0.

Although there are six functions, none of them should be very long. Also, the implementations of the last three are substantially identical except for one or two statements.

(3) Declare and implement the following class. The declaration should be made in the file machines.h, and the implementation done in the file machines.cpp.

Class **computer**:

- data members:
  - **char** array of length 21 – holds the name of the computer (up to 20 characters).
  - **pointer** to **diskDrive** – used to dynamically construct a disk drive when the computer object is constructed.
- function members:
  - Constructor. The constructor should have two parameters – a string representing the name of the computer and an integer giving the size of the disk drive. The constructor should dynamically allocate a **diskDrive** object of the specified size. It should copy the name (up to 20 characters into the **char** array data member.
  - Destructor. A destructor is necessary to de-allocate the disk drive because the actual **diskDrive** object will have been created with **new**.
  - A function called **print**. The print function should print the computer's name and have the **diskDrive** object print its information. Of course, the format should make the information easy to read.

(4) Use the main function that is posted online at http://www.eecs.northwestern.edu/~iraicu/teaching/EECS211/code/main.cpp. This main function will be used only for this assignment.

Requirements and Specifications:

(1) The argument to the **diskDrive** constructor is the number of bytes the user wants the disk to hold. This may not be a multiple of 64. On the other hand, our disk object will store data in blocks of 64 bytes. Therefore, you need to round up when computing the number of blocks. For example, if the user requested 500 bytes, the number of blocks would be 500/64, which in real arithmetic is 7 13/16. Of course, we can't have a fractional block. Therefore, your constructor would round up to the next integer, in this case 8, making a disk slightly larger than was requested. Similarly, you will need to round up when computing the number of bytes needed to store the bit map. See the **Comments…** section for an easy way to round up in integer division.

(2) Block numbers should count from 0. For example, if the drive has 50 blocks, you should refer to blocks 0-49.

(3) The **diskDrive** constructor should use **malloc** to get the memory needed for the bit map and the data storage. The destructor should use **free** to return this to the system. The prototypes for these functions are

(void *) malloc(int *number_of_byte s*);
and void free((void *) *address_to_be_freed*);

They are declared in the built-in C++ header stdlib.h. On the other hand, the **computer** constructor must use the C++ built-in operator **new**, and therefore the destructor for this class must use the C++ **delete** operator. (Note, most compilers require the .h in the #include statement for stdlib!)

(4) The **print** functions in both classes should format and label the data so that it is easy and pleasant to read. Further, the bit map should be printed in hexadecimal notation, while all other numeric data should be printed in decimal notation. You can change the radix for printing by using the special tokens **dec** and **hex** from **iostream**. For example, the statement "cout << hex;" changes all integer output to be hexadecimal until a corresponding insertion using **dec** is encountered. You may print all of the bytes in the bit map even though some of the bits in the last byte may not correspond to blocks on the drive.

(5) Remember that you need to check the block number for being valid in many of the functions. Be sure to print out informative messages that include the nature of the error and the function in which the error occurred. For example, a message "Bad block number." is not sufficient. A more informative message could look like
"Block error in allocateBlock: argument = 21, blocks on this device = 20."

(6) The constructor for the **computer** class can assume that the name will not be more than 20 characters long and therefore does not need to check for the name being too long. The minimum disk size is 128 bytes (not realistic, but good enough for our project). The

constructor must check the parameter specifying the disk size. If it is less than 128, the constructor should create a disk of size 128 bytes.

Comments, suggestions, and hints:

(1) Just a reminder that it is quite common to write a "throw away" test driver. For testing an individual class in isolation from the other parts of the project, as is the case in this assignment, the throw-away approach can be used effectively. For testing groups of interacting classes or for simulating real-time or event-driven systems, the command-line approach that we will develop in programs 4 and 5 can be very effective.

(2) The main function posted on Blackboard tests all the functionalities of your classes. As usual, I am suggesting that you build your classes in stages and only implement a few of the functions at a time. Therefore, you should comment out the sections of our test main function that call functions which you have not yet added to the classes or implemented. After you have debugged one set of functions and begin developing another set, you can un-comment the relevant lines in main.

(3) Don't attempt to write all the functions at once. Here is a good order for implementing the public interface functions:

1. Implement the constructor, destructor, and print functions for **diskDrive** first. You need the constructor function to completely create the object, and you can use the print function to verify that your next batch of functions are working correctly.
2. The functions that allocate, free and check blocks could be implemented next. They are relatively simple, and in fact their bodies are very similar to each other.
3. Finally, implement and test the **computer** class.

(4) Rounding up in integer division is quite simple. If you are dividing by k by n, just add n-1 before dividing. That is, instead of "k/n" use "(k+n-1)/n". A few examples will convince the reader that this works. Suppose we are dividing by 8, as you would do in computing the number of bytes in the bit map. One divided by 8 (1/8) is a fraction less than 1, but it should round up to 1. If we add 7 (n-1) to the numbers in the range 1-8 we get numbers in the range 8-15. Each one of these will give 1 when divided by 8 in integer arithmetic. Similarly, adding 7 to numbers in the range 25-32 gives numbers in the range 32-39, each of which results in 4 when divided by 8 in integer arithmetic.

(5) You can use **memset** for initializing blocks of storage (for example, initializing the bit map to all zeroes) and **memcpy** to copy bytes (for example, to copy the name of the computer in the **computer** class constructor). The prototypes for these functions are

void memset(char **destination*, char *data*, int *number_of_bytes*);
and void memcpy(char **destination*, char **source*, int *number_of_bytes*);

They are declared in the C++ built-in header file string.h.

(6) Bit manipulation is a very common part of computer science applications, especially in areas like operating systems, data packing and data encryption. C++ and other modern languages provide bit operations that facilitate the coding of such applications. These operations include *and*, *or*, *not*, and others. Many applications, including our program 3 assignment, make use of bit masks – a byte or word with all 0 bits except one bit is 1 or all 1 bits except one bit is 0. The *shift* and *not* operations help generate such masks. Let us consider the bit map for our **diskDrive** class. The allocated/free map is a sequence of bytes, each byte containing 8 bits. Suppose we wanted to set bit 13 (counting from 0) in the map. Bit 13 is the sixth bit in the byte 1 of the map. (Bits 0-7 are in byte 0, bits 8-15 in byte 1, etc.). Thus, 13/8 gives the byte position in the map (counting from 0, as in arrays), and 13%8 (i.e., the remainder) gives the bit position within that byte. To set that bit, i.e., make it to be 1, without interfering with the other bits in that byte, we can *or* it with a mask that has a 1 in bit 6 and 0 in all the other bits. Pictorially:

$$\begin{array}{ll} & b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 \\ or & 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0 \\ & \text{----------} \\ & b_7\ 1\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 \end{array}$$

To clear a bit we would use the bitwise *not* of that above mask and the *and* operator.

$$\begin{array}{ll} & b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 \\ and & 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \\ & \text{----------} \\ & b_7\ 0\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 \end{array}$$

To test if that bit was on or off we would *and* with the original mask and see if the result was 0 or not. The C++ book has a good section on bitwise operators and arithmetic. Here is a quick summary of what you need.

- The bitwise *and*, *or*, and *not* operator symbols are &, |, and ~ respectively. Note that the first two are like their logical test counterparts except that they use only one symbol instead of two (& instead of &&, for example). These operators can be used with C++ types char and int.
- The left and right shift operators are indicated by << and >> respectively. To generate a mask with a 1 in bit position j and zeroes in all other bits use:
  mask = 1<<j;
- To generate a mask with a 0 in bit position j and ones in all other positions use:
  mask = ~(1<<j);