Chapter 6

Strings and Files

Data processed by computers (and humans) may be classified generally into three categories: numeric data, text data, and binary data. Numeric data consists of integer and floating-point values. Text data consists of character strings and arrays of characters. Binary data is used to store graphic images, sound files, compressed and encrypted files, *etc.* In C++, numeric data is processed using numeric types such as `int` and `float`, and text data is processed using C-strings and the Standard C++ string class.

6.1 C-STRINGS

A *C-string* is an array of chars. We call them "C-strings" because that is the standard type used for processing text in the C programming language (which is a subset of C++).

EXAMPLE 6.1 Using **C-Strings**

```
char s[20] = "ABCDEFG";  // s is a C-string
cout << s << endl;       // C-strings can be output like simple types
s[4] = '*';              // C-strings are arrays
cout << s << endl;
cin >> s;                // C-strings can be input like simple types
cout << s << endl;
```

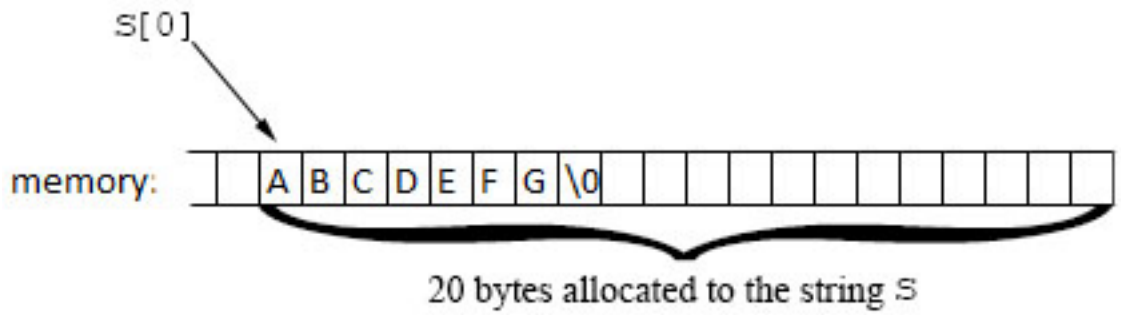
As a C-string, `s` can be input and output like a fundamental type (`int, char, float,` *etc.*) using the extraction and insertion operators `>>` and `<<`. And as an array, `s` can be manipulated using the subscript operator `[]`.

Here is a sample run of this code (with the input shown in **boldface**):

```
ABCDEFG
ABCD*FG
    Hi, Mom!
Hi,
```

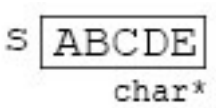
Note that the extraction operator `>>` ignores whitespace (blanks, tabs, newlines, *etc.*) that precedes the input string, and it stops extracting characters as soon as it encounters a whitespace character. That's why only the three characters `"Hi,"` were read into `s.` The rest of the input `"Mom!"` is still in the input buffer.

The declaration of `s` allocates 20 consecutive bytes in memory to the object named `s`. The initialization of `s` sets the first seven of those characters to the capital letters specified and then it sets the eighth character to 0 (called the *null character*). This can be viewed as



When the assignment `s[4] = '*';` executes, the system uses `s` to locate the starting byte (containing `'A'`) and then adds 4 to its memory address to find the byte represented by `s[4]` (which contains `'E'`). Then it replaces the `'E'` with the `'*'`.

Although it is important to remember that a C-string is really just a segment of memory, it is usually easier to think of a C-string as a primitive object, as shown at right. Note that the actual type for C-strings is `char*` which means "pointer to a `char`"; *i.e.*, a memory address.



C-strings are like fundamental objects (ints, floats, *etc.*). We think of them as objects which can be initialized and output as atomic units like ints and floats. But unlike fundamental objects, C-strings have the following non-fundamental properties:

1. A C-string is an array of `char`s that are accessible using the subscript operator: `s[i]`.
2. A C-string object has type `char*`.
3. The input operator `>>` can be used only if the string contains no whitespace.
4. C-strings cannot be assigned.

EXAMPLE 6.2 Passing a C-String to a Function

This function returns the number of capital letters that are in the C-string passed to it:

```
int caps(char* s)
{   int c=0;
    while (*s++)
        if(*s >= 'A' && *s <='Z') ++c;
    return c;
}
```

Note that the C-string type `char*` must be used in the parameter list.

The loop is controlled by the compact expression (`*s++`). The value of the expression `*s` is the actual character that `s` points to in memory. Initially, that value is the first character of the string. The effect of the postincrement operator is to advance the pointer to the next byte in memory. So on the second iteration, the value of `*s` is the second character in the string. Remember that the `char` type is actually an integer type which can act as a `bool` with zero meaning `false` and non-zero meaning `true`. So when a `char` is used as a boolean condition, the value is true except when that `char` is the null character. Since every C-string ends with the null character, the loop will continue iterating until it comes to the end of the string. This construct `while(*s++)` is the standard method for traversing a C-string.

Inside the loop, the current character `*s` is compared to see if it lies in the range of capital letters. If it does, the counter `c` is incremented. Here is a test driver:

```
int main()
{   char s[] = "404 Oak Street, SW, Tulsa, OK, USA";
    cout << "caps(" << s << ") = " << caps(s) << endl;
}
```

The output is

```
caps(404 Oak Street, SW, Tulsa, OK, USA) = 10
```

6.2 THE <cstring> LIBRARY

C++ inherits from C a collection of special functions that work on C-strings. To use them,

```
#include <cstring>
```

in your program. (If you are using a pre-Standard C++ compiler, #**include**<string.h> instead.)
Here are the declarations of five of the more useful C-string functions:

`strcat()`	`char* strcat(char* s1, const char* s2);` Appends `s2` to `s1`. Returns `s1`.
`strchr()`	`char* gtrchr(const char* s, int c);` Returns a pointer to the first occurrence of `c` in `s`. Returns `NULL` if `c` is not in `s`.
`strcmp()`	`int strcmp(const char* s1, const char* s2);` Compares `s1` with substring `s2`. Returns a negative integer, zero, or a positive integer, according to whether `s1` is lexicographically less than, equal to, or greater than `s2`.
`strcpy()`	`char* strcpy(char* s1, const char* s2);` Replaces `s1` with `s2`. Returns `s1`.
`strlen()`	`size_t strlen(const char* s);` Returns the length of `s`, which is the number of characters beginning with `s[0]` that precede the first occurrence of the `NULL` character.

EXAMPLE 6.3 Lexicographic Comparisons

Character strings are ordered alphabetically, as in a dictionary. This is called the *lexicographic ordering*. The `strcmp()` function is used to compare C-strings.

```
char* si = "pear";
char* s2 = "peach";
int cmp = strcmp(s1, s2);
cout << "cmp = " << cmp << endl;
if(cmp < 0) cout << s1 << " < " << s2 << endl;
```

```
else if(cmp == 0) cout << s1 << " == " << s2 << endl;
else cout << s1 << " > " << s2 << endl;
```

The word "pear" comes after the word "peach" in the dictionary because at the left-most letter where they differ (the fourth letter), "pear" has an "r" and "peach" has a "c". Therefore, the call `strcmp(s1, s2)` returns a positive integer:

```
cmp = 15
pear > peach
```

Note in these examples that a C-string can be defined using the **`char`*** type directly. The following three declarations are equivalent:

```
char* const s = "pear";
char s[] = "pear";
char s[5] = "pear";
```

Each declares `s` to be a constant pointer to a `char`, allocates 5 consecutive bytes in memory to `s`, copies the five characters `'p', 'e', 'a', 'r'`, and `'\0'` (the null character) to them, and then stores the address of the first byte into `s`. The declaration

```
char* s = "pear";
```

is also the same, except that this `s` is not constant; it can be incremented, as in the expression `*s++`.

EXAMPLE 6.4 Copying C-Strings with the `strcpy()` Function

C-strings cannot be assigned. Instead, the `strcpy()` function is used to copy one string to another.

```
int main()
{   char s1[] = "Beetchoven";
    char s2[] = "Bartok";
    cout << "s1 = [" << s1 << "] , strlen(s1) = " << strlen(s1) << endl;
    cout << "s2 = [" << s2 << "] , strlen(s2) = " << strlen(s2) << endl;
    strcpy(s1, s2);
    cout << "s1 = [" << s1 << "] , strlen(s1) = " << strlen(s1) << endl;
    cout << "s2 = [" << s2 << "] , strlen(s2) = " << strlen(s2) << endl;
}
```

The output is

```
S1 = [Beethoven] , strlen(sl) = 9
s2 = [Bartok] , strlen(s2) = 6
s1 = [Bartok] , strlen(sl) = 6
s2 = [Bartok] , strlen(s2) = 6
```

Note that the call `scrcpy (s1, s2)` copies the second string `s2` into the first string `s1`; like an assignment, the action is from right to left**.**

This example also illustrates the `strlen()` function which return the length of the string, not counting its null character.

6.3 FORMATTED INPUT

Rccall the idea of a stream in C++ as a conduit through which data passes (page 27). Input passes through an `istream` object and output passes through an `ostream` object. The `istream` class defines the behavior of objects like `cin`**.** The most common behavior is the use of the *extraction operator* `>>` (also called the *input operator*). It has two operands: the `istream` object from which it is extracting characters, and the object to which it copies the corresponding value formed from those characters. This process of forming a typed value from raw input charactcrs is called *formatting.*

EXAMPLE 6.5 The Extraction Operator >> Performs Formatted Input

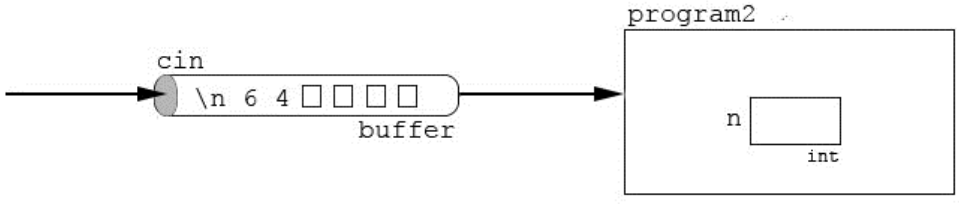
Suppose the code

```
int n;
cin >> n;
```

executes on the input

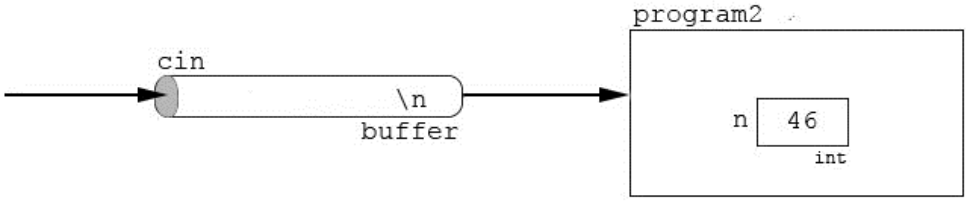
```
46
```

This input contains 7 characters: `' ', ' ', ' ', ' ', '4', '6'` and `'\n'` (4 blanks, a 4, a 6, and the newline character). It could be viewed as coming through the input stream. The stream object `cin` scans characters one at a time. If the first character it sees is a whitespace charactcr (a blank, a tab, a newline, *etc.)* it extracts it and ignores it. It continues to extract and ignore the characters in the stream until it encounters a non-whitespace character. In this example, that would be the `'4'`. Since the second operand of the expression `cin >> n` has type `int`, the `cin` object is looking for digits to form an integer. So after "eating" any preceding whitespace, it expects to find one of the 12 characters `'-','+','1','2','3','4','5','6','7','8', or '9'`. If it encounters any of the other 244



characters, it will fail. In this case, it sees the '4'. So it extracts it and then continues, expecting more digits. As long as it encounters only digits, it continues to extract them. As soon as it sees a non-digit, it stops, leaving that non-digit in the stream. In this case, that means that cin will extract exactly 6 characters: the 4 blanks, the '4', and the '6'. It discards the 4 blanks and then combines the ' 4 ' and the ' 6 ' to form the integer value 46. Then it copies that value into the object n.

After that extraction has finished, the newline character is still in the input stream. If the next input statement is another formatted input, ihen like all whitespace characters that newline character will be ignored.



The extraction operator >> *formats* the data that it receives through its input stream. This means that it extracts characters from the stream and uses them to form a value of the same type as its second operand. In the process it ignores all whitespace characters that precede the characters it uses. A direct consequence of this rule is that it is impossible to use the extraction operator to read whitespace characters. For that you must use an unformatted input function. The operator expression

```
cin >> x
```

has a value that can be interpreted in a condition as boolean; *i.e.*, either `true` or `false` depending upon whether the input is successful. That allows such an expression to be used to control a loop.

EXAMPLE 6.6 Using the Extraction Operation to Control a Loop

```
int main()
{   int n;
    while(cin >> n)
        cout << "n = " << n << endl;
}
```

Here is a sample run (with the input shown in boldface):

```
46
n = 46
22   44   66   88
n = 22
n = 44
n = 66
n = 88
33,  55,  77,  99
n = 33
```

The loop continues iterating as long as the integer data is separated by only whitespace. The first non-whitespace character, the comma ',' causes the input to fail, thereby stopping the loop.

6.4 UNFORMATTED INPUT

The <iostream> files defines several functions inputting characters and C-strings that do not skip over whitespace. The most common are the cin.get() function for reading individual characters and the cin.getline() function for reading C-strings.

EXAMPLE 6.7 Inputting Characters with the cin.get() Function

```
while(cin.get(c))
{   if(c >= 'a' && c <= 'z') c += 'A' - 'a'; // capitalize c
    cout.put(c);
    if(c == '\n') break;
}
```

This loop is controlled by the input expression (cin.get(c)). When the input stream object cin detects the end-of-file (signaled interactively by <Ctrl-D> or <Ctrl-Z>), the expression evaluates to **false** and stops the loop. This loop also terminates with a **break** statement after reading and processing the newline character '\n'. The **if** statement simply capitalizes all lowercase letters, and the cout.put(c) statement prints the character.

Here is a sample run:

```
Cogito, ergo sum?
COGITO, ERGO SUM!
```

EXAMPLE 6.8 Inputting C-Strings with the cin.getline() Function

This program shows how to read text data line-by-line into an array of C-strings:

```
const int LEN=32;     // maximum word length
const int SIZE=10;    // array size
typedef char Name[LEN]; // defines Name to be a C-string type
int main()
{   Name king[SIZE];  // defines king to be an array of 10 names
    int n=0;
    while(cin.getline(king[n++], LEN) && n<SIZE)
        ;
    --n;              // now n == the number of names read
    for(int i=0; i<n; i++)
        cout << '\t' << i+1 << ". " << king[i] << endl;
}
```

The object `king` is an array of 10 object of type `Name` which is defined to be a synonym for C-strings that hold up to 32 `chars` (31 non-null character). The function call `cin.getline (king [n++] , LEN)` reads characters from `cin` until either it has extracted `LEN-1` characters or it encounters the newline character, whichever comes first. It copies these characters into the C-string `king[n]`. If it encounters the newline character, it extracts it and ignores it (*i.e.*, it does not copy it into the C-string). Then it increments `n`. Note that the body of the `while` loop is empty. The loop stops when either `cin` detects the end-of-file or when `n == SIZE`. Since `n` starts at `0` and is incremented after the last name is read, its value is always 1 greater than the number of names read. So it gets decremented once at the end so that its value equals the number of names read. Then it is easy to print them or process them in other ways using a simple `for` loop.

When input was read from a text file that looks like this:

```
Kenneth II (971-995)
Constantine III (995-997)
Kenneth III (997-1005)
Malcolm II (1005-1034)
Duncan I (1034-1040)
Macbeth (1040-1057)
Lulach (1057-1058)
Malcolm III (1058-1093)
```

the output was

```
1. Kenneth II (971-995)
2. Constantine III (995-997)
3. Kenneth III (997-1005)
4. Malcolm II (1005-1034)
5. Duncan I (1034-1040)
6. Macbeth (1040-1057)
7. Lulach (1057-1058)
8. Malcolm III (1058-1093)
```

6.5 THE `string` TYPE

Standard C++defines a `string` type in the `<string>` file. Objects of type `string` can be declared and initialized in several ways:

```
string s1;                  // s1 contains 0 characters
string s2 = "New York";     // s2 contains 8 characters |
string s3(60, '*');         // s3 contains 60 asterisks
string s4 = s3;             // s4 contains 60 asterisks
string s5(s2, 4, 2);        // s5 is the 2-character string "Yo"
```

If the `string` is not initialized, like `s1` here, then it represents the empty string containing 0 characters. A `string` can be initialized the same way a C-string is, like `s2` here. Or a `string` can be initialized to hold a given number of the same character, like `s3` here which holds 60 stars. Unlike a `C`-string, `C++ string` objects can be initialized with a copy of another existing `string` object, like `s4` here, or with a substring of an existing string, like `s5`. Note that the standard substring designator has three parts: the parent string (`s2`, here), the starting character (`s2[4]`, here), and the length of the substring (2, here).

Formatted input works the same way for `C++ strings` as it does for `C`-strings: preceding whitespace is skipped, and input is halted at the end of the first whitespace-terminated word. `C++ strings` have a `getline()` function that works almost the same way as the `cin.getline()` function for C-strings:

```
string s = "ABCDEFG";
```

```
getline(cin, s);     // reads the entire line of characters into s
```

They also use the subscript operator the same way that C-strings do:

```
char c = s[2] ;              // assigns 'C' to c
s[4] = '*';                  // changes s to "ABCD*FG"
```

Note that the array index always counts how many characters precede the indexed character. C++ strings can be converted to C-strings like this:

```
const char* cs = s.c_str(); // converts s into the C-string cs
```

The c_str() function has return type **const char***.

The C++ string class also defines a length() function that can be used like this to determine how many characters are stored in a string:

```
cout << s.length () << endl; // prints 7 for the string s == "ABCD*FG"
```

C++ strings can be compared using the relational operators like fundamentals types:

```
if(s2 < s5) cout << "s2 lexicographically precedes s5\n";
while (s4 == s3) //...
```

You can also concatenate and append strings using the + and += operators:

```
string s6 = s + "HIJK";  // changes s6 to "ABCD*FGHIJK"
s2 += s5;                // changes s2 to "New YorkYo"
```

The substring() function is used like this:

```
s4 = s6.substr(5,3);     // changes s4 to "FGH";
```

The erase() and replace() function work like this:

```
s6.erase(4, 2);            // changes s6 to "ABCDGHIJK"
s6.replace(5, 2, "xyz");   // changes s6 to "ABCDGxyzJK"
```

The find() function returns the index of the first occurrence of a given substring:

```
string s7 = "Mississippi River basin";
cout << s7.find("si") << endl; // prints 3
cout << s7.find("so") << endl; // prints 23, the length of the string
```

If the find() function fails, it returns the length of the string it was searching.

EXAMPLE 6.9 Using the Standard C++ string Type

This code adds a nonsense syllable after each "t" that precedes a vowel. For example, it changes the sentence

```
The first step is to study the status of the C++ Standard.
```

into the sentence:

```
The first stegep is tego stegudy the stegatus of the C++ Stegandard.
```

It uses an auxiliary boolean function named is_vowel():

```
string word;
int k;
while(cin >> word)
{   k = word.find("t") + 1;
    if(k < word.length() && is_vowel(word[k]))
    word.replace(k, 0, "eg");
    cout << word << ' ';
}
```

The while loop is controlled by the input, terminating when the end-of-file is detected. It reads one word at a time. If the letter t is found and if it is followed by a vowel, then eg is inserted between that t and the vowel.

6.6 FILES

File processing in C++ is very similar to ordinaiy interactive input and output because the same kind of stream objects are used. Input from a file is managed by an ifstream object the same way that input from the keyboard is managed by the istream object cin. Similarly, output to a file is managed by an ofstream object the same way that output to the monitor or printer is managed by the ostream object cout. The only difference is that ifstream and ofstream objects have to be declared explicitly and initialized with the external name of the file which they manage. You also have to **#include** the <fstream> file (or <fstream.h> in pre-Standard C++) that defines these classes.

EXAMPLE 6.10 Capitalizing All the Words in a Text File

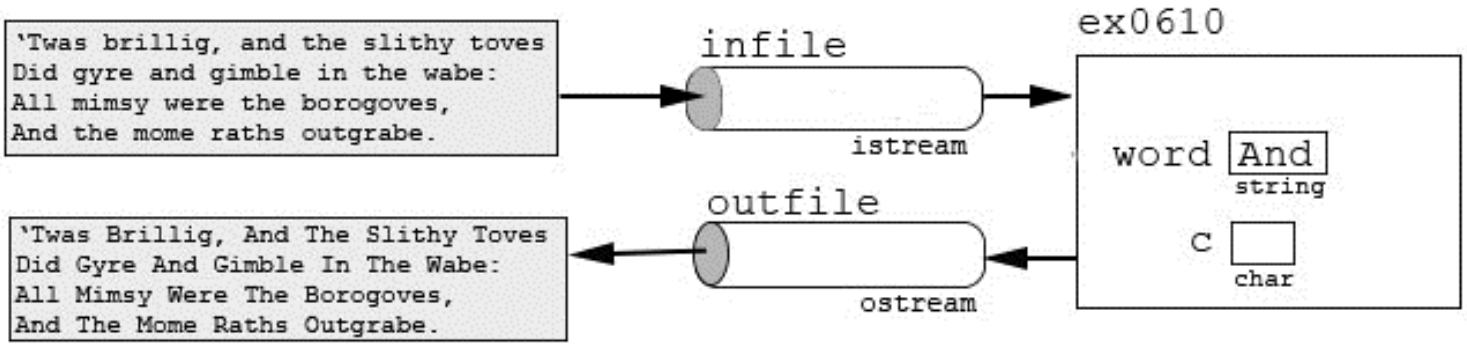
Here is a complete program that reads words from the external file named `input.txt`, capitalizes them, and then writes them to the external file named `output.txt`:

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{   ifstream infile("input.txt");
    ofstream outfile("output.txt");
    string word;
    char c;
    while(infile >> word)
    {   if (word[0] >= 'a' && word[0] <= 'z')
            word[0] += 'A' - 'a';
        outfile << word;
        infile.get(c);
        outfile.put(c);
    }
}
```

The picture below illustrates the process. Compare this with the picture on page 27.

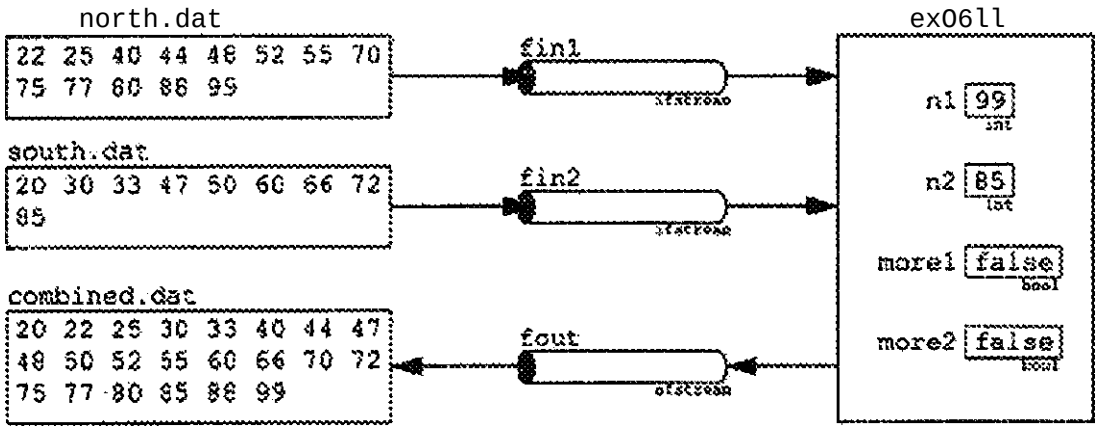
Notice that the program has four objects: an `ifstream` object named `infile`, an `ofstream` object named `outfile`, a `string` object named `word`, and a `char` object named `c`.

The advantage of using external files instead of command line redirection is that there is no limit to the number of different files that you can use in the same program.



EXAMPLE 6.11 Merging Two Sorted Data Files

This program merges two files into a third file. The numbers stored in the files north. dat and south. dat are sorted in increasing order. The program reads these two input files simultaneously and copies all their data to the file c omb i ne d. da t so that they are all together in increasing order:



```
bool more(ifstream& fin, int& n)
{   if(fin >> n) return true;
    else return false;
}
bool copy(ofstream& fout, ifstream& fin, int& n)
{   fout << " " << n;
    return more(fin, n);
}
int main()
{   ifstream fin1("north.dat");
    ifstream fin2("south.dat");
    ofstream fout("combined.dat");
    int n1, n2;
    bool more1 = more(fin1, n1);
    bool more2 = more(fin2, n2);
    while(more1 && more2)
    {   if(nl < n2) more1 = copy(fout, fin1, n1);
        else more2 = copy(fout, fin2, n2);
    }
    while(more1)
        more1 = copy(fout, fin1, n1);
    while(more2)
        more2 = copy(fout, fin2, n2);
    fout << endl;
}
```

The `more()` function is used to read the data from the input files. Each call attempts to read one integer from the `fin` file to the reference parameter `n`. It returns `true` if it is successful, otherwise `false`. The `copy()` function writes the value of `n` to the **`fout`** file and then calls the `more()` function to read the next integer from the `fin` file into `n`. It also returns `true` if and only if it is successful.

The first two calls to the `more()` function read 22 and 20 into `nl` and `n2`, respectively. Both calls return true which allows the main while loop to begin. On that first iteration, the condition `(nl < n2)` is `false`, so the `copy()` function copies 20 from `n2` into the `combined.dat` file and then calls the `more()` function again which reads 30 into `n2`. On the second iteration, the condition `(nl < n2)` is true (because 22 < 30), so the `copy()` function copies 22 from `nl` into the `combined.dat` file and then calls the `more()` function again which reads 25 into `nl`. The next iteration writes 25 to the output file and then reads 40 into `nl`. The next iteration writes 30 to the output file and then reads 33 into `n2`. This process continues until 85 is written to the output file from `n2` and the next call to `more()` fails, assigning `false` to `more2`. That stops the main `while` loop. Then the second `while` loop iterates three times, copying the last three integers from `north.dat` to `combined.dat` before it sets `morel` to `false`. The last loop does not iterate at all.

Note that file objects (`finl, fin2, fout`) are passed to function the same way any other objects are passed. However, they must always be passed by reference.

6.7 STRING STREAMS

A *string stream* is a stream object that allows a string to be used as an internal text file. This is also called *in-memory I/O.* String streams are quite useful for buffering input and output Their types `istringstream` and `ostringstream` are defined in the `<sstream>` file.

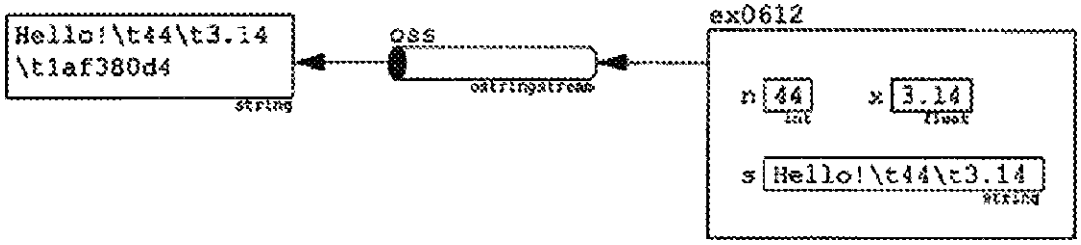
EXAMPLE 6.12 Using an Output String Stream

Here is a complete Standard C++ program:

```
#include <iostream>
#include <sstream>
#include <cstring>
using namespace std;
int main()
{   ostringstraam oss;
    int n = 44;
    float x = 3.14;
```

```
    oss << "Hello!\t" << n << '\t' << x;
    string s = oss.str();   // copies the stream's string to s
    oss << '\t' << &n;
}
```

The objects in this program can be visualized like this:



The stream object's anonymous string is drawn outside the program to emphasize its natural analogy with an external file. (See the drawings on pages 27 and 116.) But actually, both the stream object and its anonymous string are objects in the program.

The object `oss` is an output string stream. It serves as a conduit to an anonymous string which can be read with the built-in `oss.str()` function that is bound to the `oss` object. The insertion operator `<<` is used to insert the string literal `"Hello!\t"`, the integer `n`, the character `'\t'`, and the `float x`. Then the contents of `oss`'s anonymous string are copied to the local string `s`. Then the character `'\t'` and the address of the object `n` are inserted into `oss`.

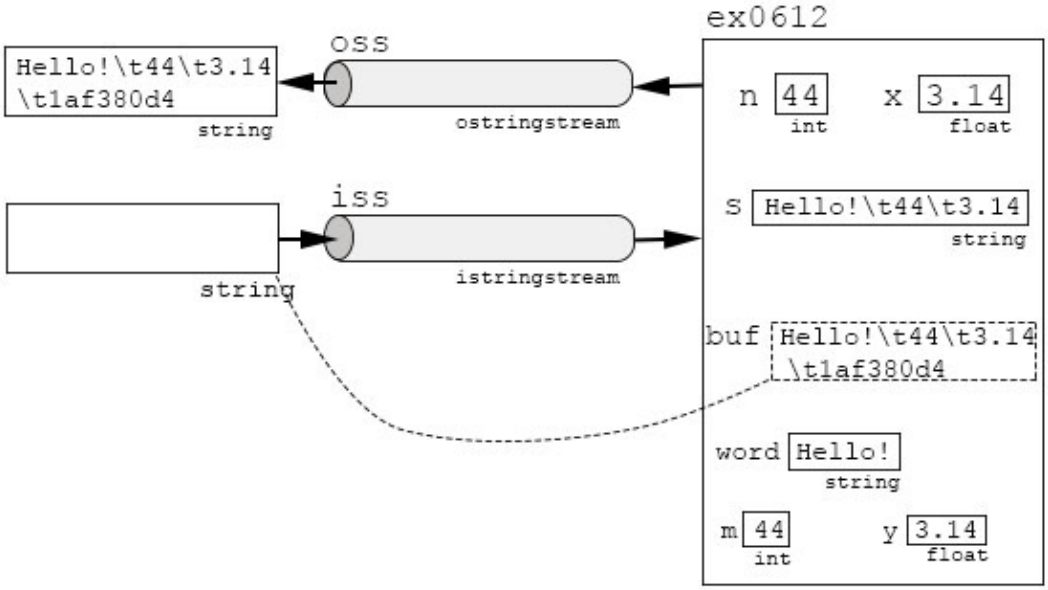
EXAMPLE 6.13 Using an Input String Stream

Suppose that we append the following lines of code to the program in Example 6.12:

```
const string buffer = oss.str();
istringstrean iss(buffer);   // binds the stream iss to the string
string word;
int m;
float y;
iss >> word >> m >> y;
```

The first line copies the current contents of `oss`'s string into `buffer`. Then the input string stream iss is defined and bound to `buffer`. This means that all extractions from `iss` will come from the contents of `buffer`, just as though it were an external text file. Then after declaring another `string, int,` and `float`, their values are read from `iss`.

The objects in this expanded program can be visualized like this:



Note that the contents of buffer can be accessed two ways: as elements of a string, or by formatted input through the iss object.

```
char c = buffer[16]; // assigns 'f' to c
iss >> word;  // copies "1af380d4" into word
```

6.8 RANDOM ACCESS FILES

Files can be accessed directly like an array. This is called direct access or random access. The access location is set by using the file's `seekg()` function for reading and its `seekp()` function for writing.

EXAMPLE 6.14 Random Access of a Text File

This complete program creates a dummy text file and then allows the user to select a random location to rewrite a part of it.

```
#include <cstdlib>    // defines the exit() function
#include <fstream>
#include <iostream>
#include <cstring>
void load(fstream& f)
{   for(int i=l; i <= 5; i++)       // write 5 64-char lines
    {   for(int j=l; j <= 60; j++)
            f << j % 10;            // write 60 digits
        f << " " << i << '\n';      // write  4  chars
    }
}
void dump(fstream& f)
{   f.seekg(0);                 // start at the beginning of the file
    char c;
    while(f.get(c))             // echo each character in f to cout
    cout.put(c);
    f.clear();       // resets the file's eofbit
}
int main()
{   fstream file("Demo.dat", ios::in | ios::out);
    if(!file)
    {   cerr << "File Demo.dat could not be opened.\n";
        exit(1);
    }
    load(file);
    dump(file);
    int pos, len;
    cin >> pos >> len;
    file.seekp(pos);
    string s(len, '*');
    file.write(s.c_str(), s.length());
    dump(file);
}
```

The first line creates an external text file named `Demo.dat`. It is bound to an `fstream` object named `file`. Since we want to read and write to `Demo. dat` using the same stream object, we declare it to have type `fstream` (instead of `ifstream` or `ofstream`) and initialize it with the `ios::in | ios::out` flag (meaning that it is open for both input and output).

File processing in general is very prone to error. There are many kinds of run-time errors that can happen to programs that use files. For example, the system may be unable to create the new file because your directory is full or its file permissions prevent it. So it is recommended that whenever a file is opened (by declaring its file stream object), it should be checked for possible error before proceeding with its processing. This is usually done with an `if` statement like the one on lines 2-4 of the `main()` function here. The condition `(!file)` means that the file is not ready for processing. If so, we simply print an error message and quit the program. The `exit()` function is defined in the `<cstdlib>` file. Passing the integer `1` is simply a signal to the operating system that the program is terminating with an error.

If the file is ready for processing, we call the `load()` function to fill it with 5 lines of digits and then the `dump()` function to display the complete file on the screen. Then the program interactively reads a position number `pos` and a length number `len`. In the run shown below, we entered 200 for the position and 17 for the length.

The next three statement change the 17 characters in positions 201-217 to asterisks. The call `file.seekp(pos)` function moves its write pointer (the "p" in "`seekp`" stands for "put") to position `pos`; *i.e.*, it advances that many characters past the beginning of the file. Then `s` is defined to be a string of 17 asterisks. The call `file.write(cs, len)` replaces the file's `len` characters that are located by its write pointer with the same number of characters in the C-string `cs`. So the call

```
file.write(s.c_str(), s.length());
```

changes the next 17 digits to asterisks. Notice the use of the string's `c_str()` and `length()` functions.

Here is a sample run:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890  1
12345678901234567890123456789012345678901234567890123456789012345678901234567890  2
12345678901234567890123456789012345678901234567890123456789012345678901234567890  3
12345678901234567890123456789012345678901234567890123456789012345678901234567890  4
12345678901234567890123456789012345678901234567890123456789012345678901234567890  5
200 17
12345678901234567890123456789012345678901234567890123456789012345678901234567890  1
12345678901234567890123456789012345678901234567890123456789012345678901234567890  2
12345678901234567890123456789012345678901234567890123456789012345678901234567890  3
12345678*****************67890123456789012345678901234567890123456789012345678901  4
12345678901234567890123456789012345678901234567890123456789012345678901234567890  5
```

The arrangement of the (background) digits makes it easy to check that the correct 17 characters were "asterisked out." The call to the `seekp()` function specified that writing begins after the 200th character. Since each line contains 64 characters (counting the newline character), the first 3 lines contain 3 – 64 = 192 characters, which means that the 200th character must be the 8th character of line 4. So the 9th – 25th digits on line 4 are replaced with asterisks.

The `seekp(pos)` and `write(cs, len)` functions are used to write the `C`-string `cs` of length `len` to a random access file at position `pos`**.** In the same way, the `seekg(pos)` and `read(cs, len)` functions are used to read the `len` characters starting at position `len` from a random access files into the `C`-string `cs.`

EXAMPLE 6.15 Processing Student Grades

Consider the two data files shown below. The `Students.dat` file contains students records, one per line, in four fields: the student's id number, name, total credits earned, and grade point average. The `Grades.dat` file contains grade records, one per line, in three fields: a student's id number, final grade in a course, and credit hours for that course. The following program processes these grades, updating the student records by adding the new credits and recomputing the grade point averages of those students whose ids appear in the `Grades.dat` file. Note that the records in both files are sorted by student id. Also in both files, each field is separated by a single tab character `'\t'` and each record is terminated by the newline character `'\n'`.

```
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream.h>
using namespace std;
int credit(string grade_rec);
void get(string& student_rec, fstream& students, string id);
void update(string& student_rec, char grade, int credit);
void put(string student_rec, fstream& students);
const int STUDENT_REC_SIZE=30;
int main()
{   fstream students("Students.dat", ios::in | ios::out);
    ifstream grades("Grades.dat");
    if(!students || !grades)
    {   cerr << "One of the files could not be opened.\n";
        exit(1);
    }
    string grade_rec;   // a line from the Grades.dat file
    string student_rec; // a line from the Students.dat file
    string _id; // the student id from the grade_rec
    char _grade;              // the student grade from the grade_rec
    int _credit;              // the student credit from the grade_rec
    while (getline(grades, grade_rec))
    {   _id = grade_rec.substr(0,9);
        _grade = grade_rec[10];
        _credit = credit(grade_rec);
        get(student_rec, students, _id);
        update(student_rec, _grade, __credit);
        put(student_rec, students);
    }
}
```

Students.dat

```
017142031     Vance,  Vera    85   3.06
027910908     Nixon,  Nora    47   2.61
104148606     Allen,  Adam    29   2.84
118229053     Tyson,  Tara    37   2.80
129830779     Evans,  Earl    92   3.21
370221320     Ogden,  Owen    80   3.23
391235525     Chang,  Carl    87   3.10
407890754     Gomez,  Gary    50   2.35
413114410     Rosen,  Raul    73   2.18
487072864     Davis,  Dora    43   2.39
490130095     Ukrop,  Urey    20   3.75
569844817     Singh,  Sara    19   3.63
645378014     Paine,  Perl    51   3.00
678206512     Jones,  Judy    37   2.77
709404115     Hanes,  Hope    94   3.44
724978457     Brown,  Bill    63   2.17
861480354     Frost,  Fred    17   1.63
863050853     Russo,  Rose    95   1.99
907723489     Levin,  Lisa    65   2.53
926311364     Irvin,  Ivan    76   2.97
956733958     Moore,  Mark    25   3.01
966739174     Knopp,  Karl    41   1.88
```

Grades.dat

```
027910908     B     4
104148606     A     3
118229053     B     3
370221320     F     3
407890754     B     4
487072864     C     3
490130095     C     4
569844817     A     3
645378014     A     4
678206512     B     3
709404115     A     3
724978457     C     3
926311364     A     4
956733958     D     3
```

The `Students.dat` file is opened for both input and output because its records are to be updated. Each iteration of the main loop reads one `graderec` record from the `grades` file. It extracts the student `_id` and `grade` directly from the string, and uses the `credit()` function to get the numeric credit value from the string. Then it calls the `get()` function to get the `student_rec` record from the students file that matches the `id`. That record is then updated by the `update()` function and written back to the `students` file by the `put()` function.

Here is the `credit()` function:

```
int credit(string grade_rec)
{   string s = grade_rec.substr(12, 1);
    istringstream ss(s);
    int n;
    ss >> n;
    return n;
}
```

The credit integer is a single digit occupying the 13th character of the `graderec`. That is extracted as a substring, and then the input string stream `ss` is used to take advantage of the automatic formatting of the extraction operator » to get the integer value into `n` so it can be returned. Here is the `get()` function:

```
void get (strings student_rec, fstreamS students, string id)
{   while(!students.eof())
    {   getline(students, student_rec);
        if(student_rec.substr(0,9) == id) break;
    }
}
```

This continues reading records from the student_rec file until it finds the one that has the same id as that passed in to the function. Notice that it uses the string `substr()` function to extract the 9-character id from each student record.

Here is the `update()` function:

```
void update(string& student_rec, char grade, int credit)
{   int grade_points = 4 - int(grade - 'A');
    grade_points = (grade_points < 0 ? 0 : grade_points);
    string s = student_rec.substr(22,7);
    istringstream iss(s);
    int credits;
    iss >> credits;
    float gpa, points;
    iss >> gpa;
    points = credits*gpa + credit*grade_points;
    credits += credit;
    gpa = points/credits;
    ostringstream oss;
```

```
        oss << credits << '\t' << setprecision(3) << gpa;
        s = oss.str();
        student_rec.replace(22, 7, s.substr(0,7));
    }
```

First it converts the letter grade into its numeric equivalent: 4 for an 'A', 3 for a `'B'`, 2 for a `'C'`, 1 for a `'D'` and 0 for an `'F'`. Then it uses the input string stream `iss` to extract the numeric `credits` and `gpa` from the student record. These lie in columns 23-29 of the `student_rec`. It computes the student's new `credits` and `gpa`. Then it uses the output string stream `oss` to insert the new `credits` and `gpa` into the string `s`. Finally it uses the string `substr()` and `replaced` functions to insert these new values back into the `student_rec`.

Here is the `put()` function:

```
void put(string student_rec, fstream& students)
{   student_rec += string(1, '\n'); // append the newline character
    const char* p = student_rec.c_str();
    int location = students.tellg();
    location -= STUDENT_REC_SIZE;
    students.seekp(location);
    students.write(p, STUDENT_REC_SIZE);
}
```

The `student_rec` contains 29 characters. Each record in the Students.dat file is 30 characters long, the last being the newline character. So we have to append `'\'` to `student_rec` before writing it back to the file. Then we can update the file by re-writing the record. The `write()` function requires the equivalent `C`-string `p` to be passed.

Review Questions

6.1 What is the difference between a C-string and a C++ string?

6.2 What is the difference between formatted input and unformatted input?

6.3 Why can't whitespace be read with the extraction operator?

6.4 What is a stream?

6.5 How does C++ simplify the processing of strings, external files, and internal files?

6.6 What is the difference between sequential access and direct access?

6.7 What do the `seekg()` and `seekp()` functions do?

6.8 What do the `read()` and `write()` functions do?

Problems

6.9 Describe what the following code does:

```
char cs1[] = "ABCDEFGHIJ";
char cs2[] = "ABCDEFGH";
cout << cs2 << endl;
cout << strlen(cs2) << endl;
cs2[4] = 'X';
if(strcmp(csl, cs2) < 0) cout << csl << " < " << cs2 << endl;
else cout << csl << " >= " << cs2 << endl;
char buffer[80];
strcpy(buffer, csl);
strcat(buffer, cs2);
char* cs3 = strchr(buffer, 'G');
cout << cs3 << endl;
```

6.10 Describe what the following code does:

```
string s = "ABCDEFGHIJKLMNOP";
cout << s << endl;
cout << s.length() << endl;
s[8] = '!';
s.replace(8, 5, "xyz");
s.erase(6, 4);
cout << s.find("!");
cout << s.find("?");
cout << s.substr(6, 3);
s += "abcde";
```

```
string part(s, 4, 8);
string stars(8, '*');
```

6.11 Describe what happens when the code

```
string s;
int n;
float x;
cin >> s >> n >> x >> s;
```

executes on each of the following inputs:

a. `ABC 456 7.89 XYZ`
b. `ABC 4567 .89 XYZ`
c. `ABC 456 7.8 9XYZ`
d. `ABC456 7.8 9 XYZ`
e. `ABC456 7 .89 XYZ`
f. `ABC4 56 7.89XY Z`
g. `AB C456 7.89 XYZ`
h. `AB C 456 7.89XYZ`

6.12 Trace the execution of the merge program in Example 6.11 on page 117 on the following two data files:

north.dat

```
27 35 38 52 55 61 81 87
```

south.dat

```
31 34 41 45 49 56 63 74
92 95
```

Show each value of the variables `nl`, `n2`, `morel`, and `more2`, as they change.

Programming Problems

6.13 Write a program that reads full names, one per line, and then prints them in the standard telephone directory format. For example, the input

```
Johann Sebastian Bach
George Frederic Handel
Carl Phillipp Emanuel Bach
Joseph Haydn
Johann Christian Bach
Wolfgang Amadeus Mozart
```

would be printed as:

```
Bach, Johann S.
Handel, George F.
Bach, Carl P. E.
Haydn, Joseph
Bach, Johann C.
Mozart, Wolfgang A.
```

6.14 Write a program that counts and prints the number of lines, words, and letter frequencies in its input. For example, the input:

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
```

would produce the output:

```
The input had 5 lines, 37 words, and the following
letter frequencies:
     A: 10   B: 3   C: 2   D: 13  E: 15  F: 1   G: 3   H: 4
     I: 7    J: 0   K: 1   L: 8   M: 0   N: 12  0: 20  P: 0
     Q: 0    R: 11  S: 5   T: 11  U: 3   V: 3   W: 6   X: 0
     Y: 2    Z: 0
```

6.15 Implement and test the following function:

```
void reduce(strings s);
// Changes all capital letters in s to lowercase
```

```
// and removes all non-letters from the beginning and end.
// EXAMPLE: if s == "'Tis,", then reduce(s) makes it "tis"
```

Hint: First write and test the following three boolean functions:

```
bool is_uppercase(char c);
bool is_lowercase(char c);
bool is_letter(char c);
```

6.16 Modify your program from Problem 6.14 so that it counts the frequencies of words instead of letters. For example, the input

```
[I] then went to Wm. and Mary college, to wit in the spring of 1760, where
I continued 2 years. It was my great good fortune, and what probably fixed
the destinies of my life that Dr. Wm. Small of Scotland was then professor
of Mathematics, a man profound in most of the useful branches of science,
with a happy talent of communication, correct and gentlemanly manners,
& an enlarged & liberal mind. He, most happily for me, became soon attached
to me & made me his daily companion when not engaged in the school; and
from his conversation I got my first views of the expansion of science
& of the system of things in which we are placed,
```

would produce the output

```
The input had 11 lines and 120 words, with the following frequencies:

          i:   3         then:  2             went:  1
         to:   3           wm:  2              and:  4
       mary:   1      college:  1              wit:  1
         in:   4          the:  6           spring:  1
         of:  11            :  6            where:  1
  continued:   1        years:  1               it:  1
        was:   2           my:  3            great:  1
       good:   1      fortune:  1             what:  1
    probably:   1        fixed:  1         destinies:  1
       life:   1         that:  1               dr:  1
      small:   1     scotland:  1         professor:  1
mathematics:   1            a:  2               man:  1
   profound:   1         most:  2            useful:  1
   branches:   1      science:  2              with:  1
      happy:   1       talent:  1     communication:  1
    correct:   1  gentlemanly:  1           manners:  1
         an:   1     enlarged:  1           liberal:  1
       mind:   1           he:  1           happily:  1
        for:   1           me: 11            became:  1
       soon:   1     attached:  1              made:  1
        his:   2        daily:  1         companion:  1
       when:   1          not:  1           engaged:  1
     school:   1         from:  1      conversation:  1
        got:   1        first:  1             views:  1
  expansion:   1       system:  1            things:  1
      which:   1           we:  1               are:  1
     placed:   1
```

6.17 Write a program that right-justifies text. It should read and echo a sequence of left-justified lines and then print them in right-justified format. For example, the input

```
Listen, my children, and you shall hear
Of the midnight ride of Paul Revere,
On the eighteenth of April, in Seventy-five;
Hardly a man is now alive
Who remembers that famous day and year.
```

would be printed as

```
                    Listen, my children, and you shall hear
                       Of the midnight ride of Paul Revere,
                 On the eighteenth of April, in Seventy-five;
                                 Hardly a man is now alive
                     Who remembers that famous day and year.
```

6.18 Implement and test the following function:

```
string Roman(int n);
// Returns the Roman numeral equivalent to the Hindu-Arabic
// numeral n.
// PRECONDITIONS: n > 0, n < 3888
// EXAMPLES: Roman(1776) returns "MDCCLXXVI",
// Roman(1812) returns "MDCCCXII", Roman(1945) returns "MCMXLV"
```

6.19 Implement and test the following function:

```
int HindArabic(string s);
// Returns the Hindu-Arabic numeral equivalent to the Roman
// numeral given in the string s.
// PRECONDITIONS: s contains a valid Roman numeral
// EXAMPLES: HindArabic("MDCCLXXVI") returns 1776,
// HindArabic("MDCCCXII") returns 1812
```

Note that this is the inverse of the `Roman()` function in Problem 6.18.

Hint: Write an auxiliary function `int v(string s, int i)` that returns the digit for the Roman numeral character `s[i]`; e.g., `v("MDCCCXII", 1)` returns `500`.

6.20 Implement Algorithm 1.4 on page 6 to convert decimal numerals to hexadecimal:

```
string hexadecimal(int n);
// Returns the hexadecimal numeral that represents n.
// PRECONDITION: n >= 0
// POSTCONDITION: each character in the returned string is a
// hexadecimal digit and that string is the dexadecimal
// equivalent of n
// EXAMPLE: hexadecimal(11643) returns "2d7b"
```

Hint: Write an auxiliary function `char c(int k)` that returns the hexadecimal character for the hexadecimal digit `k`; *e.g.*, `c(14)` returns `'e'`.

6.21 Implement Algorithm 1.4 on page 6 to convert hexadecimal numerals to decimal:

```
int decimal(string s);
// Returns the decimal numeral that represents the hexadecimal
// numeral stored in the string s.
// PRECONDITION: s.length() > 0 and each s[i] is a hexadecimal
// digit
// POSTCONDITION: the returns value is the decimal equivalent
// EXAMPLE: decimal("2d7b") returns 11643
```

Note that this is the inverse of the `hexadecimal()` function in Problem 6.20.

Hint: Write an auxiliary function `int v(string s, int i)` that returns the decimal digit for the hexadecimal character `s[i]`; e.g., `v("2d7b", 3)` returns `12`.

6.22 Implement and test the following function:

```
void reverse(string& s);
// Reverses the string s.
// POSTCONDITION: s[i] <--> s[len-i-l]
// EXAMPLE: reverse(s) changes s = "ABCDEFG" into "GFEDCBA"
```

Hint: Use a temporary string.

6.23 Implement and test the following function:

```
bool is_palindrome(string s);
// Returns true iff s is a palindrome
// EXAMPLES: is_palindrome("RADAR") returns true,
// is_palindrome("ABCD") returns false
```

6.24 Modify the program in Example 6.11 on page 117 so that it merges the following two files of sorted lines of text, writing the resulting sorted lines both to a file named `Presidents.dat` and to `cout`:

`Democrats.dat`

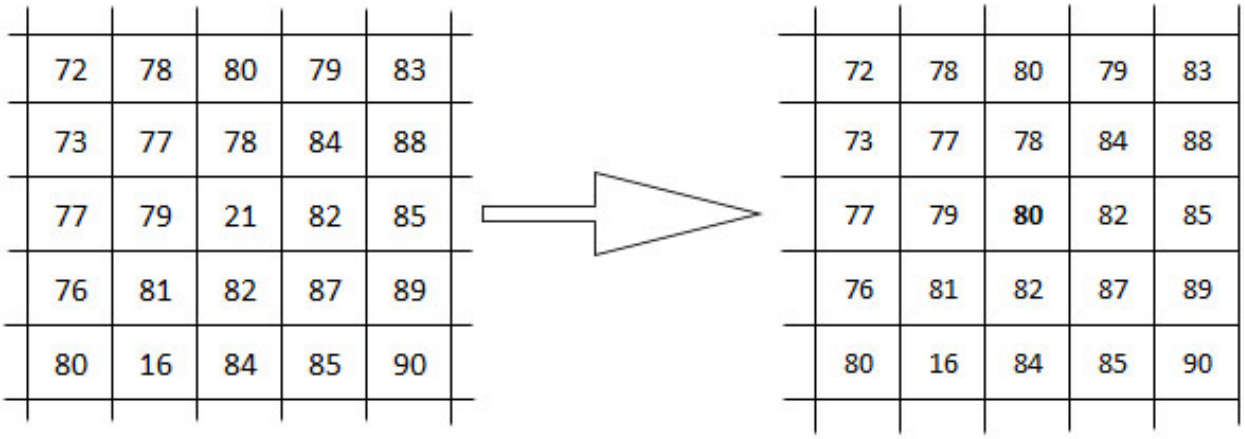
```
Carter, James Earl
Clinton, William Jefferson
Johnson, Lyndon Baines
Kennedy, John Fitzgerald
Roosevelt, Franlin Delano
Truman, Harry S
```

`Republicans.dat`

```
Bush, George Herbert Walker
Eisenhower, Dwight David
Ford, Gerald Rudolph
Nixon, Richard Milhous
Reagan, Ronald Wilson
```

Hint: Use `getline(fin, s)`.

6.25 A certain image file format stores graphic images using one byte per pixel (256 colors) It uses the first 8 bytes to store the image's dimensions: the first four-byte integers giving the number of rows and the second the number of columns. The remaining bytes represent the two-dimensional image stored in *row-major* form. For example, the first 8 bytes evaluate to the two integers 5000 and 6400, then the file will have 32,000,000 more bytes (a 32-MB file) for its 32,000,000 pixels, arranged in 5000 rows of 6400 pixels each. Write the following function that cleans up a noisy image file by smoothing erroneous pixels. If a pixel value differs from the average value of its four neighboring pixels by more than a given tolerance, then that value is changed to the average value. For example, the call `clean(f,16)` would make the following change:




```
void clean(fstreams f, int tolerance);
// Cleans the image file by averaging singularities.
// PRECONDITIONS: f is bound to a file that contains mn+8 bytes,
// where m and n are 4-byte unsigned integers stored in the
// first 8 bytes; f is open for input and output
// POSTCONDITION: no pixel value differs from the average of its
// four neighbors by more than tolerance
```

Hint: The locations of the four neighbors of a pixel located at byte `x` are `x-n, x-1, x+1,` and `x+n` (north, west, east, and south).

6.26 Modify the program in Example 6.15 on page 121 so that assumes a 3-digit credits field (instead of 2 digits) in the `Students.dat` file. Then adjust the actual data file accordingly, change the record for `Hope Hanes` to 114 credits, and run your program.

Solutions

6.1 A C-string is an array of chars that uses the null character `'\0'` to mark the end of the string. A C++ string is an object whose `string` type is defined in the `<string>` file and which has a large repertoire of function, such as `length()` and `replace()`:

```
char cs[8] = "ABCDEFG"; // cs is a C-string
string s = "ABCDEFG";   // s is a C++ string
cout << s << " has " << s.length() << " characters.\n";
s.replace(4, 2, "yz");  // changes s to "ABCDyzG"
```

6.2 Formatted input uses the extraction operator >> which ignores whitespace. Unformatted input uses the `get()` and `getline()` functions. The `get()` function reads the next character in the input stream without ignoring whitespace. The `getline()` function reads all the rest of the characters in the input stream until it reaches the newline character `'\n'`, which it extracts and ignores.

6.3 Whitespace (blanks, tabs, newlines, *etc.*) cannot be read with the extraction operator because it ignores all whitespace.

6.4 A *stream* is an object that manages input and output between a program and a data source. C++ allows `<iostream>` objects for interactive I/O (*viz.*, `cin` and `cout`), `<fstream>` objects for external files, and `<sstream>` objects for internal files (string streams).

6.5 C++ simplifies the processing of strings, external files, and internal files, by defining the same family of functions and operations for all three. For example, the extraction operator >> works the same way for inputting a `double` from the keyboard, from an external file, or from a string stream.

6.6 Sequential access must begin at the beginning and access each element in order, one after the other. Direct access allows the access of any element directly by locating it by its index number or address. Arrays allow direct access. Magnetic tape has only sequential access, but CDs had direct access. If you are on a railroad train, to go from one car to another you must use sequential access. But when you board the train initially you have direct access. Direct access is faster than sequential access, but it requires some external mechanism (array index, file byte number, railroad platform).

6.7 The `seekg()` and `seekp()` functions position the get pointer and the put pointer, respectively, in an external file to allow direct access. For example, the call `input. Seekg(24)` positions the get pointer at byte number 24 in the file bound to the file stream named `input.`

6.8 The `read()` and `write()` functions are used for direct access input and output, respectively, of external files. For example, the call `input.read(s.c_str(), n)` would copy `n` bytes to the string `s` directly from the file bound to the file stream named `input.`

6.9
```
char cs1[] = "ABCDEFGHIJ";     // defines cs1to be that C-string
char cs2[] = "ABCDEFGH";       // defines cs1to be that C-string
cout << cs2 << endl;           // prints: ABCDEFGH
cout << strlen (cs2) << endl;  // prints: 8
cs2[4] = 'X';                  // changes cs2 to "ABCDXFGH"
if(strcmp(cs1, cs2) < 0) cout << csl << " <  " << cs2 << endl;
else cout << csl << " >= " << cs2 << endl;
                               // prints: ABCDEFGHIJ < ABCDXFGH
char buffer[80];               // defines buffer to be a C-string of < 80 chars
strcpy(buffer, cs1);           // changes buffer to "ABCDEFGHIJ"
strcat(buffer, cs2);           // changes buffer to "ABCDEFGHIJABCDXFGH"
char* cs3 = strchr(buffer, 'G'); // make cs3 point to buffer[6]
cout << cs3 << endl;           // prints: GHIJABCDXFGH
```

6.10
```
string s = "ABCDEFGHIJKLMNOP"  // defines s to be that string
cout << s << endl;             // prints: ABCDEFGHIJKLMNOP
cout << s.length() << endl;    // prints: 16
s[8] = '!';                    // changes s to "ABCDEFGH!JKLMNOP"
s.replace(10, 5, "xyz");       // changes s to "ABCDEFGH!JxyzP"
s.erase(2,4);                  // changes s to "ABGH!JxyzP"
cout << s.find("!") << endl;    // prints: 4
cout << s.find("?") << endl;    // prints: 10
cout << s.substr(3,6) << endl;  // prints: H!Jxyz
s += "abcde";     // changes s to "ABGH!JxyzPabcde"
string part(s, 1, 10);   // defines part to be "BGHlJxyzPa"
string stars(8, '*');    // defines stars to be "*********"
```

6.11 ***a.*** `ABC 456 7.89 XYZ`
Assigns `"ABC"` to `s`, 456 to `n`, 7.89 to `x`, and then `"XYZ"` to `s.`
b. `ABC 4567 .89 XYZ`
Assigns `"ABC"` to `s`, 4567 to `n`, 0.89 to `x`, and then "XYZ" to `s.`
c. `ABC 456 7.8 9XYZ`
Assigns "ABC" to `s`, 456 to `n`, 7.8 to `x`, and then "9XYZ" to `s.`
d. `ABC456 7.8 9 XYZ`
Assigns `"ABC456"` to `s`, and then crashes because 7.8 is not a valid integer literal.
e. `ABC456 7 .89 XYZ`
Assigns `"ABC456"` to `s`, 7 to `n`, 0.89 to `x`, and then "XYZ" to `s.`
f. `ABC4 5 67.89XY Z`
Assigns `"ABC 4 "` to `s`, `56` to `n`, and then crashes because `7.89XY` is not a valid `float` literal.
g. `AB C456 7.89 XYZ`
Assigns `"AB"` to `s` and then crashes because `C456` is not a valid integer literal. (Note that the hexidecimal numeral `c4 56`, which can also be written `C456`, would qualify as a valid integer literal. But on input, hexadecimal numerals must be prefixed with `"0x"`, as in `0xc456.`)
h. `AB C 456 7 . 89XYZ`
Assigns `"AB"` to `s` and then crashes because `C` is not a valid integer literal.

6.12 Tracing the merge program:

nl	n2	morel	more2
27	31	true	true
35	34		
	41		
38			
52			
	45		
	49		
	56		
55			
61			
	63		
81			
	74		
	92		
87		false	
	95		false

6.13

```
int main()
{   string word, first, last;
    char c;
    bool is_first, is_last = true;
    string name[32];
    int n=0;
    while(cin >> word)
    {   cin.get(c);   // should be either a blank or a newline
        is_first = is_last;   // current word is a first name
        is_last = bool(c == '\n'); // current word is a last name
        if(is_first) first = word;
        else if(is_last) name[n++] = word + ", " + first;
        else first += " " + word.substr(0,1) + "."; // add initial
    }
    --n;
    for(int i=0; i<n; i++)
        cout << '\t' << i+1 << ". " << name[i] << endl;
}
```

6.14

```
int main()
{   string word;
    const int SIZE=91;  // for frequency array (int('Z') == 90)
    int lines=0, words=0, freq[SIZE] = (0), len;
    char c;
    while(cin >> word)
    {   ++words;
        cin.get(c);
        if(c == '\n') ++lines;
        len = word.length();
        for(int i=0; i<len; i++)
        {   c = word[i];
            if(c >= 'a' && c <= 'z') c += 'A' - 'a'; // capitalize c
            if(c >= 'A' && c <= 'Z') ++freq[c]; // count c
        }
```

```
    }
    cout << "The input had " << lines << " lines, " << words
         << " words,\nand the following letter frequencies:\n";
    for(int i= 65; i<SIZE; i++)
    {   cout << '\t' << char(i) << ": " << freq[i];
        if(i > 0 && i%8 == 0) cout << endl;     // print 8 to a line
    }
    cout << endl;
}
```

6.15

```
bool is_upper(char c)
{   return bool(c >= 'A' && c <= 'Z');
}
bool is_lower(char c)
{   return bool(c >= 'a' && c <= 'z');
}
bool is_letter(char c)
{   return bool(is_upper(c) || is_lower(c));
}
void reduce(string& s)
{   while(s.length() > 0 && !is_letter(s[0]))
        s.erase (0,1);
    int k = s.length() - 1;
    while(k > 0 && !is_letter(s[k--]))
        s.erase(k+1, 1);
    int len = s.length();
    if(len == 0) return;
    for(int i=0; i<len; i++)
        if(is_upper(s[i])) s[i] += 'a' - 'A';
}
```

6.16

```
int main()
{   string s;
    const int SIZE=1000;      // assume at most 1000 different words
    string word[SIZE];        // holds words read
    int lines=0, words=0, n=0, freq[SIZE]={0};
    char c;
    while(cin >> s)
    {   reduce(s);
        if(s.length == 0) continue;
        ++words;
        cin.get(c);
        if(c == '\n') ++lines;      // count line
        for(int i=0; i<n; i++)
        if(word[i] == s) break;
        if(i == n) word[n++] = s;   // add word to list
        ++freq[i];                 // count word
    }
    cout << "The input had " << lines << " lines and " << words
         << " words,\nwith the following frequencies:\n";
    for(int i=0; i<n; i++)
    {   s = word[i];
        if(i > 0 && i%3 == 0) cout << endl;     // print 3 to a line
        cout << setw(16) << setiosflags(ios::right)
             << s.c_str() << ": " << setw(2) << freq[i];
    }
```

```
        cout << endl;
    }
```

6.17

```
    int main()
    {   const int SIZE=100; // maximum number of lines stored
        string line[SIZE], s;
        int n=0, len, maxlen=0;
        while(!cin.eof())
        {   getline(cin, s);
            len = s.length();
            if(len > 0) cout << s << endl;
            if(len > maxlen) maxlen = len;
            line[n++] = s;
        }
        --n;   // n == number of lines read
        for(int i=0; i<n; i++)
        {   s = line[i];
            len = s.length();
            cout << string(maxlen-len, ' ') << s << endl;
        }
    }
```

6.18

```
string Roman(int n)
{   int d3 = n/1000; // the thousands digit
    string s(d3, 'M');
    n %= 1000;
    int d2 = n/100;  // the hundreds digit
    if(d2 == 9) s += "CM";
    else if (d2 >= 5)
    {   s += "D";
        s += string(d2-5, 'C');
    }
    else if(d2 == 4) s += "CD";
    else s += string(d2, 'X');
    n %= 100;
    int d1 = n/10;   // the tens digit
    if(d1 == 9) s += "XC";
    else if(d1 >= 5)
    {   s = "L";
        s += string(d1-5, 'X');
    }
    else if(d1 == 4) s += "XL";
    else s += string(d1, 'X');
    n %= 10;
    int d0 = n/1;    // the ones digit
    if(d0 == 9) s += "IX";
    else if(d0 >= 5)
    {   s += "V";
        s += string(d0-5, 'I');
    }
    else if(d0 == 4) s += "IV";
    else s += string(d0, 'I');
    return s;
}
```

6.19

```
int v(string s, int i)
{  char c = s[i];
    if(c == 'M') return 1000;
    if(c == 'D') return 500;
    if(c == 'C') return 100;
    if(c == 'L') return 50;
```

```
    if(c == 'X') return 10;
    if(c == 'V') return 5;
    if(c == 'I') return 1;
    return 0;
}
int HindArabic(string s)
{   int n = v(s,0);
    int len = s.length();
    for(int i=l; i<len; i++)
        if(v(s, i) <= v(s,i-1)) n += v(s,i);
    else n -= 2*v(s,i-l);
    return n;
}
```

6.20

```
char c(int k)
{   assert(k >= 0 && k <= 15);
    if(k < 10) return char(k + '0');
    return char(k - 10 + 'a');
}
string hexadecimal(int n)
{   if(n == 0) return string(1, '0');
    string s;
    while(n > 0)
    {   s = string(1, c(n%16)) + s;
        n /= 16;
    }
    return s;
}
```

6.21

```
int v(string s, int i)
{   char c = s[i];
    assert(c >= '0' && c <= '9' || c >= 'a' && c <= 'f');
    if(c >= '0' && c <= '9') return int(c - '0');
    else return int(c - 'a' + 10);
}
int decimal(string s)
{   int len = s.length();
    assert(len > 0);
    int n=0;
    for(int i=0; i<len; i++)
        n = 16*n + v(s,i);
    return n;
}
```

6.22

```
void reverse (string& s)
{   string temp = s;
    int len = s.length();
    for(int i=0; i<len; i++)
        s[i] = temp[len-i-1];
}
```

6.23

```
bool is_palindrome(string s)
{   int len = s.length();
    for(int i=0; i<len/2; i++)
        if(s[i] != s[len-i-l]) return false;
    return true;
}
```

6.24

```
bool more(ifstream& fin, string& s)
{   cout << s << endl;
    fout << s << endl;
    return more(fin, s);
}
bool more(ifstream& fin, string& s)
{   if(getline(fin, s)) return true;
```

```
    else return false;
}
int main()
{   ifstream finl("Democrats.dat");
    ifstream fin2("Republicans.dat");
    ofstream fout("Presidents.dat");
    string s1, s2;
    bool more1 = more(fin1, s1);
    bool more2 = more(fin2, s2);
    while(more1 && more2)
    if(s1 < s2) more1 = copy(fout, fin1, s1);
    else more2 = copy(fout, fin2, s2);
    while(morel)
        morel = copy(fout, fin1, s1);
    while(more2)
        more2 = copy(fout, fin2, s2);
    fout << endl;
}
```

6.25

```
typedef unsigned short int Byte;
const int MAX = 4*255;    // 4* maximum value for a Byte
void clean(fstream& f, int tolerance);
{   long unsigned int m, n;     // 4-byte integers
    string pixel(1,' ');  // each pixel is a 1-character string
    string north(1,' '), west(1,' '), eastf1,' '), southf1,' ');
    int x, sum;
    Byte p, a, b, c, d, ave;
    f.seekg(0);
    f >> m >> n;   // read dimensions of image
    for(int i=l; i<m-l; i++)    // examine every interior pixel
    for(int j =1; j<n-l; j++)
    {   x = i*n + j + 8;  // the address of the current pixel
        f.seekg(x);
        f.read(pixel.c_str(), 1);
        p = Byte(pixel[0]);   // convert string character to a Byte
        f.seekg(x-n);
        f.read(north.c_str(), 1);
        a = Byte(north[0]);
        f.seekg(x-1);
        f.read(west.c_str(), 1);
        b = Byte(west[0]);
        f.seekg(x+1);
        f.read(east.c_str(), 1);
        c = Byte(east[0]);
        f.seekg(x+n);
        f.read(south.c_str(), 1);
        d= Byte(south[0]);
        sum = a + b + c + b;
        sum = (sum > MAX ? MAX : sum); // watch out for overflow
        ave = sum/4;
        if(p > ave + tolerance || p < ave - tolerance)
        {   f.seekp(x);
            pixel[0] = char(ave);
            f.write(pixel.c_str(), 1);
        }
     }
 }
```

Note: This solution is not optimized for speed.