CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

CLASS TEMPLATES

- Class templates are generally used for data storage (container) classes. Consider the following class

```
class Stack
{
private:
    int st[MAX];       //array of ints
    int top;           //index number of top of stack

public:
    Stack();           //constructor
    void push(int var); //takes int as argument
    int pop();         //returns int value
};
```

CLASS TEMPLATES

- If we wanted to store data of type long in a stack, we would need to define a completely new class:

```
class LongStack
{
private:
    long st[MAX];           //array of longs
    int top;                //index number of top of stack
public:
    LongStack();            //constructor
    void push(long var);    //takes long as argument
    long pop();             //returns long value
};
```

- Similarly, we would need to create a new stack class for every data type we wanted to store.

CLASS TEMPLATES

- It would be nice to be able to write a single class specification that would work for variables of all types, instead of a single basic type.

- As you may have guessed, **class templates** allow us to do this.

```cpp
#include <iostream>
using namespace std;
const int MAX = 100;    //size of array

template <class Type>
class Stack{

     Type st[MAX];      //stack: array of any type
     int top;           //index of top(most recent) of stack

public:
     Stack()            //constructor
     {top = -1;}

     void push(Type var) //put number on stack
     { st[++top] = var; }

     Type pop()         //take number off stack
     { return st[top--]; }
};
```

```cpp
int main(){

     Stack <float> s1;  //s1 is object of class Stack<float>
     s1.push(1111.1F); //push 3 floats, pop 3 floats
     s1.push(2222.2F);
     s1.push(3333.3F);
     cout << "1: " << s1.pop() << endl;
     cout << "2: " << s1.pop() << endl;
     cout << "3: " << s1.pop() << endl;

     Stack <long> s2;      //s2 is object of class Stack<long>
     s2.push(123123123L); //push 3 longs, pop 3 longs
     s2.push(234234234L);
     s2.push(345345345L);
     cout << "1: " << s2.pop() << endl;
     cout << "2: " << s2.pop() << endl;
     cout << "3: " << s2.pop() << endl;

     Stack <int> s3;    //s3 is object of class Stack<int>
     s3.push(123123);  //push 3 ints, pop 3 ints
     s3.push(234234);
     s3.push(345345);
     cout << "1: " << s3.pop() << endl;
     cout << "2: " << s3.pop() << endl;
     cout << "3: " << s3.pop() << endl;

return 0;
}
```

CLASS TEMPLATES

- Here the class **Stack** is presented as a template class.
- The approach is similar to that used in function templates. The `template` keyword and class `Stack` signal that the entire class will be a template.
- A template argument, named `Type` in this example, is then used (instead of a fixed data type such as int) everyplace in the class specification where there is a reference to the type of the array `st`.
    - There are three such places: the definition of `st`, the argument type of the `push()` function, and the return type of the `pop()` function
-

CLASS TEMPLATES

- Class templates differ from function templates in the way they are instantiated,
- Classes, however, are instantiated by defining an object using the template argument.

```
Stack <float> s1;
```

- This creates an object, **s1**, a stack that stores numbers of type **float**.
- The compiler provides space in memory for this object's data, using type **float** wherever the template argument **Type** appears in the class specification.
- It also provides space for the member functions. These member functions also operate exclusively on type **float**.

CLASS TEMPLATES

- Similarly, Creating a Stack object that stores objects of a different type, as in **Stack<long> s2**; creates not only a different space for data, but also a new set of member functions that operate on type **long**.

- Note that the name of the type of **s1** consists of the class name Stack *plus the template argument*: `Stack<float>`.
  - This distinguishes it from other classes that might be created from the same template, such as `Stack<int>` or `Stack<long>`.

- In the previus example, the member functions of the class template were all defined within the class. If the member functions are defined externally (outside of the class specification), we need a new syntax.

- The next program shows how this works.

```
#include <iostream>
using namespace std;
const int MAX = 100;
//////////////////////////////////////////////////
template <class Type>
class Stack
{
     Type st[MAX];
     int top;
public:
     Stack();
     void push(Type var);
     Type pop();
};
//////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack() //constructor
{
     top = -1;
}

template<class Type>
void Stack<Type>::push(Type var)
{
     st[++top] = var;
}

template<class Type>
Type Stack<Type>::pop()
{
     return st[top--];
}
```

```
int main()
{
     Stack<float> s1; //s1 is object of class Stack<float>
     s1.push(1111.1F); //push 3 floats, pop 3 floats
     s1.push(2222.2F);
     s1.push(3333.3F);
     cout << “1: “ << s1.pop() << endl;
     cout << “2: “ << s1.pop() << endl;
     cout << “3: “ << s1.pop() << endl;

     Stack<long> s2; //s2 is object of class Stack<long>
     s2.push(123123123L); //push 3 longs, pop 3 longs
     s2.push(234234234L);
     s2.push(345345345L);
     cout << “1: “ << s2.pop() << endl;
     cout << “2: “ << s2.pop() << endl;
     cout << “3: “ << s2.pop() << endl;

return 0;
}
```

C++ EXCEPTION HANDLING

- An **exception** is a problem that arises during the execution of a program.

- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as <u>an attempt to divide by zero</u>.

- Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

C++ EXCEPTION HANDLING

- Imagine an application that creates and interacts with objects of a certain class. Ordinarily the application's calls to the class member functions cause no problems.

- However, sometimes application makes a mistake, causing an error to be detected in a member functions. This member function then informs the application that an error has occurred.

- When exceptions are used, this is called ***throwing*** an exception.

- In the application we install/write a separate section of code to handle the error. This code is called an ***exception handler*** or ***catch block***, it *catches* the exceptions thrown by the member function.

C++ EXCEPTION HANDLING

- Any code in the application that uses objects of the class is enclosed in a ***try block.***
- Errors generated in the try block will be caught in the catch block, Code that doesn't interact with the class need not be in a try block

C++ EXCEPTION HANDLING

- The exception mechanism uses three new C++ keywords: **throw**, **catch**, and **try**.
  - **try** – The **try** statement allows you to define a block of code to be tested for errors while it is being executed. A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
  - **throw** – The **throw** keyword throws an exception when a problem is detected, which lets us create a custom error
  - **catch** – The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block. A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

```
try {  //protected block
     int age;
     cin>>age;
    if (age >= 18 && Age<=60 ) {
        cout << "Access granted - you are old enough.";
    }
    else {throw (age); //exception thrower}

    cin>>name;
    if (Name[0]==‘x’)
        throw(Name);
}
catch (int myNum) { //exception handler
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
catch(char* str){cout<<str<<“ is invalid name”;}
```

- We use the **try** block to test some code: If the **age** variable is less than **18**, we will **throw** an exception, and handle it in our **catch** block.

- In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

- If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

```
#include <iostream>
using namespace std;

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 2;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
     cout << msg << endl;
   }

   return 0;
}
```

HANDLE ANY TYPE OF EXCEPTIONS (...)

- If you do not know the throw type used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

```
try {
  int age = 15;
  if (age > 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least 18 years old.\n";
}
```

STREAMS IN C++

- A *stream* is a general name given to a flow of data.
- In C++ a stream is represented by an object of a particular class.
- So far we've used the **cin** and **cout** stream objects.
- Different streams are used to represent different kinds of data flow.

Advantages of Streams

- One reason is simplicity.
  - If you've ever used a **%d** formatting character when you should have used a **%f** in **printf()**, you'll appreciate this.
  - There are no such formatting characters in streams, since each object already knows how to display itself.
- Another reason is that you can overload existing operators and functions, such as the insertion (<<) and extraction (>>) operators, to work with classes that you create.

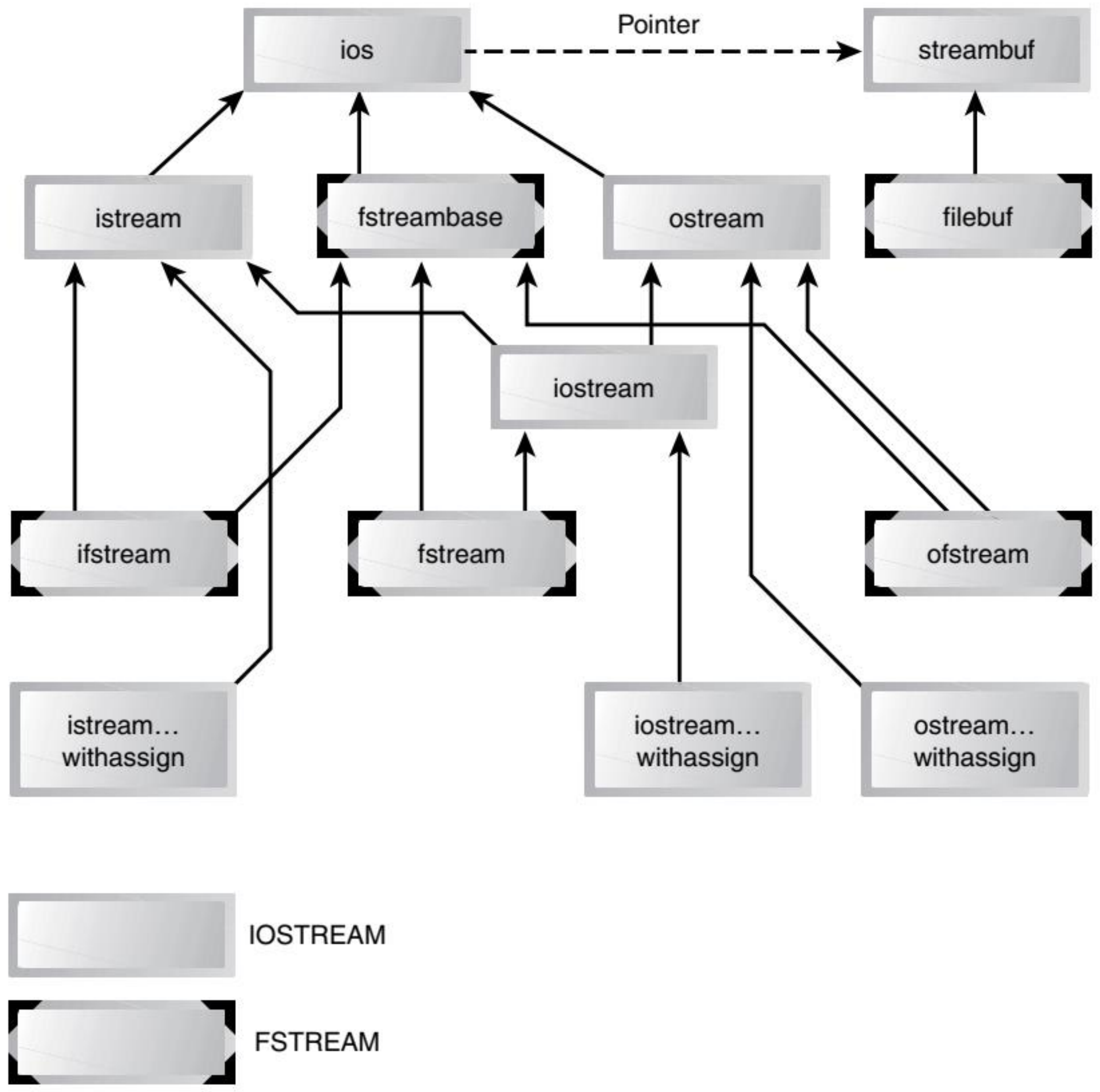

FIGURE 12.1

Stream class hierarchy.

STREAMS IN C++

- We've already made extensive use of some stream classes. The extraction operator **>>** is a member of the **istream** class, and the insertion operator **<<** is a member of the **ostream** class.
  - Both of these classes are derived from the **ios** class

- The ios class is the *granddaddy* of all the stream classes, and contains the majority of the features you need to operate C++ streams.

- **Manipulators** are formatting instructions inserted directly into a stream.
  - We've seen examples before, such as the manipulator endl, which sends a newline to the stream and flushes it:

```
cout << “To each his own.” << endl;
```

TABLE 12.2 No-Argument `ios` Manipulators

Manipulator	*Purpose*
`ws`	Turn on whitespace skipping on input
`dec`	Convert to decimal
`oct`	Convert to octal
`hex`	Convert to hexadecimal
`endl`	Insert newline and flush the output stream
`ends`	Insert null character to terminate an output string
`flush`	Flush the output stream
`lock`	Lock file handle
`unlock`	Unlock file handle

THE istream CLASS

- The **istream** class, which is derived from **ios**, performs input-specific activities, or extraction.

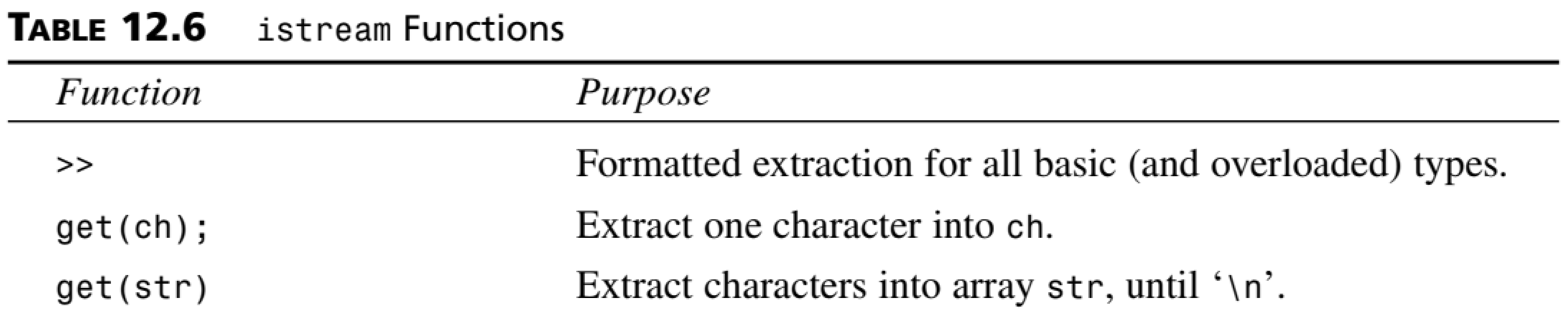

TABLE 12.6 `istream` Functions

Function	*Purpose*
`>>`	Formatted extraction for all basic (and overloaded) types.
`get(ch);`	Extract one character into `ch`.
`get(str)`	Extract characters into array `str`, until ‘`\n`’.

TABLE 12.6 Continued

Function	*Purpose*
`get(str, MAX)`	Extract up to `MAX` characters into array.
`get(str, DELIM)`	Extract characters into array `str` until specified delimiter (typically ‘`\n`’). Leave delimiting char in stream.
`get(str, MAX, DELIM)`	Extract characters into array `str` until `MAX` characters or the `DELIM` character. Leave delimiting char in stream.
`getline(str, MAX, DELIM)`	Extract characters into array `str`, until `MAX` characters or the `DELIM` character. Extract delimiting character.
`putback(ch)`	Insert last character read back into input stream.
`ignore(MAX, DELIM)`	Extract and discard up to `MAX` characters until (and including) the specified delimiter (typically ‘`\n`’).
`peek(ch)`	Read one character, leave it in stream.
`count = gcount()`	Return number of characters read by a (immediately preceding) call to `get()`, `getline()`, or `read()`.
`read(str, MAX)`	For files—extract up to `MAX` characters into `str`, until `EOF`.
`seekg()`	Set distance (in bytes) of file pointer from start of file.
`seekg(pos, seek_dir)`	Set distance (in bytes) of file pointer from specified place in file. `seek_dir` can be `ios::beg`, `ios::cur`, `ios::end`.
`pos = tellg(pos)`	Return position (in bytes) of file pointer from start of file.

THE ostream CLASS

- The **ostream** class handles output or insertion activities.

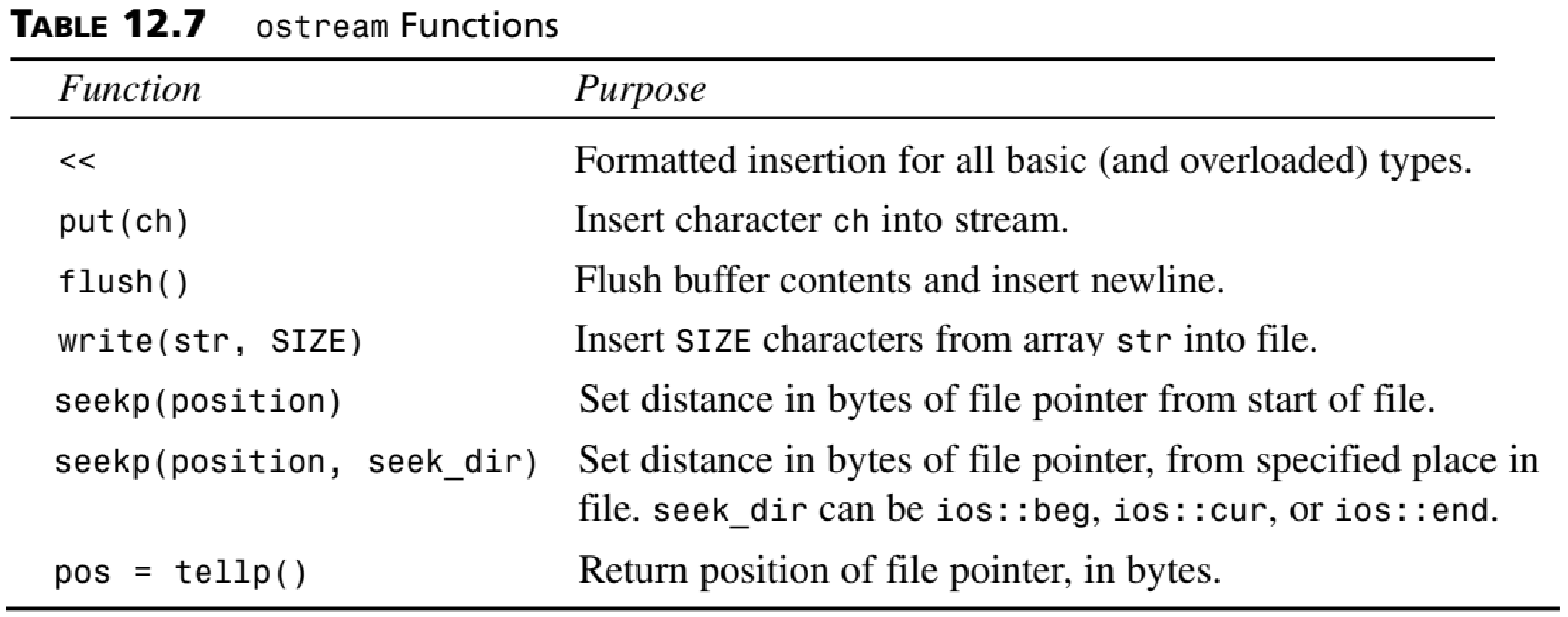

TABLE 12.7 `ostream` Functions

Function	*Purpose*
`<<`	Formatted insertion for all basic (and overloaded) types.
`put(ch)`	Insert character `ch` into stream.
`flush()`	Flush buffer contents and insert newline.
`write(str, SIZE)`	Insert `SIZE` characters from array `str` into file.
`seekp(position)`	Set distance in bytes of file pointer from start of file.
`seekp(position, seek_dir)`	Set distance in bytes of file pointer, from specified place in file. `seek_dir` can be `ios::beg`, `ios::cur`, or `ios::end`.
`pos = tellp()`	Return position of file pointer, in bytes.

DISK FILE I/O WITH STREAMS (FILING)

- Most programs need to save data to disk files and read it back in.

- Working with disk files requires another set of classes: **ifstream** for input, **ofstream** for output and **fstream** for both input and output.

- Objects of these classes can be associated with disk files, and we can use their member functions to read and write to the files .

WRITING DATA (TO DISK FILES)

```
#include <fstream>                  //for file I/O
#include <iostream>
#include <string>
using namespace std;

int main(){
    char ch = 'x';
    int j = 77;
    double d = 6.02;
    string str1 = "Kafka";
    string str2 = "Proust";

    ofstream outfile(“fdata.txt"); //create ofstream object

    outfile<< ch   << j << ' ' << d<< str1 << ' '<< str2;

    cout << "File written\n";

    return 0;
           }
```

READING DATA (FROM DISK FILE)

```
#include <fstream>                      //for file I/O
#include <iostream>
#include <string>
using namespace std;

int main(){
        char ch;
        int j;
        double d;
        string str1;
        string str2;
        ifstream infile("fdata.txt");    //create ifstream object

        infile >> ch >> j >> d >> str1 >> str2;
        cout << ch << endl << j << endl<< d << endl<< str1 << endl<< str2 << endl;
        return 0;
        }
```

STRINGS WITH EMBEDDED BLANKS

- The technique of our last examples won't work with char* strings containing embedded blanks.

- To handle such strings, you need to write a specific delimiter character after each string, and use the **getline()** function, rather than the extraction operator, to read them in.

- To extract the strings from the file, we create an **ifstream** and read from it one line at a time using the **getline()** function, which is a member of **istream**. This function reads characters, including whitespace, until it encounters the '\n' character, and places the resulting string in the buffer supplied as an argument

```
#include <fstream>
using namespace std;
int main()
{
    ofstream outfile("TEST.TXT");
    outfile << "I fear thee, ancient Mariner!\n";
    outfile << "I fear thy skinny hand\n";
    outfile << "And thou art long, and lank, and brown,\n";
    outfile << "As is the ribbed sea sand.\n";

return 0;
}
```

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    const int MAX = 80;                 //size of buffer
    char buffer[MAX];                   //character buffer

    ifstream infile("TEST.TXT");        //create file for input

    while( !infile.eof() )              //until end-of-file
    {
        infile.getline(buffer, MAX);    //read a line of text
        cout << buffer << endl;         //display it
    }
return 0;
}
```

CHARACTER I/O

- The **put()** and **get()** functions, which are members of **ostream** and **istream**, respectively, can be used to output and input single characters

```
#include <fstream>                     //for file functions
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string str = "Time is a great teacher, but unfortunately "
                "it kills all its pupils.  Berlioz";

   ofstream outfile("TEST.TXT");      //create file for output
   for(int j=0; j<str.size(); j++)    //for each character,
      outfile.put( str[j] );          //write it to file
   cout << "File written\n";
   return 0;
   }
```

```
int main()
   {
   char ch;                               //character to read
   ifstream infile("TEST.TXT");           //create file for input
   while( infile )                        //read until EOF or error
      {
      infile.get(ch);                     //read character
      cout << ch;                         //display it
      }
   cout << endl;
   return 0;
   }
```

CLOSING FILES

- So far in our example programs there has been no need to close streams explicitly because they are closed automatically when they go out of scope; this invokes their destructors and closes the associated file.
- However, if both the output stream os and the input stream is
  are associated with the same file, EDATA.DAT, the first stream must be closed before the second
  is opened. We use the close() member function for this