I/O Streams -- Other Features

`eof()` member function

- A useful member function of the input stream classes is `eof()`
  - stands for **end of file**
  - returns a `bool` value, answering the question "Are we at the end of the file?" (or is the "end-of-file" character the next one on the stream?)
  - Can be used to indicate whether the end of an input file has been reached, when reading sequentially
- Very useful when reading files where the size of the file or the amount of data to be read is not known in advance

```
while (!in1.eof())    // while not at the end of the file
{
   // read and process some input from the file
}
```

- While useful for files, can also be used with `cin`, where the user types a key combination representing the "end-of-file" character
  - On Unix and Mac systems, type *`ctrl-d`* to enter the end-of-file character
  - On Windows, type *`ctrl-z`* to enter the end-of-file character
- count.cpp -- An example that reads in a file consisting of any number of integers, using `eof()` to decide when to stop, then computes sum and average.
  - A sample input file that this program will handle

Character I/O

Character Output

- We've already used the insertion operator to print characters:

```
char letter = 'A';
cout << letter;
```

- There is also a member function (of output stream classes) called **`put()`**, which can be used to print a character. It's prototype is:

```
ostream& put(char c);
```

  Sample calls:

```
char ch1 = 'A', ch2 = 'B', ch3 = 'C';
```

```
cout.put(ch1);        // equivalent to:  cout << ch1;
cout.put(ch2);        // equivalent to:  cout << ch2;
```

Since it returns type `ostream&`, it can be cascaded, like the insertion operator:

```
cout.put(ch1).put(ch2).put(ch3);
```

- Note: The `put()` function doesn't really do anything more special than the insertion operator does. It's just listed here for completeness

Character Input

- There are many versions of the extraction operator >>, for reading data from an input stream. This includes a version that reads characters:

```
char letter;
cin >> letter;
```

  - However, Consider this example, which attempts to copy an input file to an output file, character by character (using the extraction operator).
    What happens? Why?
  - Remember, all *built-in* versions of the extraction operator for input streams will ignore *leading white space* by default
- Here are the prototypes of some other useful member functions (of input stream classes) for working with the input of characters:

```
int peek();                  // returns next char, doesn't extract
int get();                   // extracts one char from input, returns ascii value
istream& get(char& ch);      // extracts one char from input, stores in ch
```

- **`peek()`** -- this function returns the ascii value of the *next* character on the input stream, but does **not** extract it
- **`get()`** -- the two `get` functions both extract the next single character on the input stream, and they do **not** skip any white space.
  - The version with no parameters returns the ascii value of the extracted character
  - The version with the single parameter stores the character in the parameter, passed by reference. Returns a reference to the stream object (or 0, for end-of-file)
- Examples:

```
char ch1, ch2, ch3;
cin >> ch1 >> ch2 >> ch3;     // reads three characters, skipping white space

// no parameter version of get -- no white space skipped
ch1 = cin.get();
ch2 = cin.get();
ch3 = cin.get();

// Single-parameter version of get, which can also be cascaded
cin.get(ch1).get(ch2).get(ch3);
```

```
// example of peek() -- trying to read a digit, as a char
char temp = cin.peek();       // look at next character
if (temp < '0' || temp > '9')
   cout << "Not a digit";
else
   ch1 = cin.get();           // read the digit
```

- copy.cpp -- Here is a *good* version of the file-copy program, which uses `get()` to read the characters

- Other useful input stream functions:
  - **`ignore()`** member function - skips either a designated number of characters, or skips up to a specified delimiter. Examples:

```
cin.ignore();          // skip the next 1 character of input
cin.ignore(30);        // skip the next 30 characters of input
cin.ignore(100,'\n');  // skip to the next newline, up to 100 characters maximum
                       //  i.e. skip no more than 100
```

  - **`putback()`** member function - puts a character back into the input stream
- Additional code examples, illustrating:
  - ignore
  - putback
  - peek

Useful character functions (library `cctype`)

The C library called **`cctype`** contains many useful character functions. They are not specifically geared towards I/O -- they are just useful for working with characters.

Here's a quick description of some of the useful functions. See the chart on page 247 of the textbook for more details and examples.
All of these functions take a single character as a parameter -- assume that `ch` is a `char`:

- `toupper(ch)` -- returns the uppercase version of ch (if it's a letter).
- `tolower(ch)` -- returns the lowercase version of ch (if it's a letter).
- `isupper(ch)` -- returns true if `ch` is an uppercase letter, false otherwise
- `islower(ch)` -- returns true if `ch` is a lowercase letter, false otherwise
- `isalpha(ch)` -- returns true if `ch` is a letter of the alphabet, false otherwise
- `isdigit(ch)` -- returns true if `ch` is a digit ('0' through '9'), false otherwise
- `isalnum(ch)` -- returns true if `ch` is a letter *or* a digit, false otherwise
- `isspace(ch)` -- returns true if `ch` is a white space character, false otherwise
- There are a few more, but these are the most commonly-used ones

Passing Stream Objects into Functions

- In a function prototype, any *type* can be used as a formal parameter type or as a return type.
  - This includes *classes*, which are programmer-defined types
- Streams can be passed into functions as parameters (and/or returned).
  - Because of how the stream classes were set up, they can only be passed *by reference*, however
- So, for instance, the following can be return types or parameter types in a function:
  - `ostream&`
  - `istream&`
  - `ofstream&`
  - `ifstream&`
- **Why?** -- functions that do output can be written that are more versatile, by allowing the output to go to a variety of places
- Example of a more limited function:

```
void Show()
{
  cout << "Hello, World\n";
}
```

  A call to this function **always** prints to standard output (`cout`)

```
Show();
```

- Same function, more versatile:

```
void Show(ostream& output)
{
  output << "Hello, World\n";
}
```

  Notice that I can do the printing to different output destinations now:

```
Show(cout);           // prints to standard output stream
Show(cerr);           // prints to standard error stream
```

- This works with file stream types, too:

```
void PrintRecord(ofstream& fout, int acctID, double balance)
{
  fout << acctID << balance << '\n';
}
```

  Now I could call this function to print the same data format to different files:

```
ofstream out1, out2;
out1.open("file1.txt");
out2.open("file2.txt");
PrintRecord(out1, 123, 45.67);         // prints to file1.txt
PrintRecord(out1, 124, 67.89);         // prints to file1.txt
PrintRecord(out2, 1000, 123.09);       // prints to file2.txt
PrintRecord(out2, 1001, 2087.64);      // prints to file2.txt
```

Inheritance and Streams

- The stream classes are related to each other through a feature called *inheritance*

    - class `ifstream` inherits all the features of `istream`, which is why they are similar
    - class `ofstream` inherits all the features of `ostream`, which is why they are similar
    - We might say that `istream` is the *parent* of class `ifstream` (and `ostream` is the parent of `ofstream`)
- This pertains to function parameters in that variables of the *child* type may be passed in where the *parent* type is expected (but not vice versa)
- Examples:

```
// prototypes
void PrintRecord(ofstream& fout, int id, double bal);
void PrintRecord2(ostream& out, int id, double bal);

// calls
ofstream out1;
out1.open("file.txt");

PrintRecord(out1, 12, 34.56);   // legal
PrintRecord(cout, 12, 34.56);   // NOT legal (attempt to pass parent into child type)

PrintRecord2(out1, 12, 34.56);  // legal
PrintRecord2(cout, 12, 34.56);  // legal (pass child into parent)
```

- **Conclusion**: Using the parent type for your parameters is more versatile!