Chapter 9 C++ Classes

Objectives

- To write non-dynamic memory C++ classes.
- To learn how to use the built-in C++ string class.
- To learn how to read and write ASCII files in C++.
- To learn how to overload operators in C++ as methods and as functions.
- To learn how to write class variables and methods in C++.

9.1 Basic Syntax and Semantics

The reasons for and benefits of using classes in C++ are the same as they are in Python. Classes allows us to encapsulate the data and methods for interacting with the data into one syntactic unit. Data hiding allows programmers to use the class without worrying about or understanding the internal implementation details. If the programmer using the class only calls the methods for interacting with the data and does not directly change the instance variables, we are assured that the data integrity of our class is maintained (i.e., assuming the class implementation is correct, manipulating the class through the methods will not result in inconsistent data in the class instance). Classes also make it easier to reuse the code in more than one application. This section covers the basic syntax and semantics of C++ classes. We will examine some of the more advanced class topics in later sections and later chapters.

Before we start examining the syntax for C++ classes, we will discuss some of the terminology differences between Python and C++. Python officially calls members

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

of a class *attributes*; attributes can be either variables or functions. Python has a built-in function named `getattr` that stands for "get attribute" and is used to access the attributes of a class. If we have an instance `r` of the `Rational` class defined in section 2.5, the following two statements are equivalent:

```
print r.num
print getattr(r, 'num')
```

Note that the `getattr` function takes an object and a string and returns the attribute specified by the string for the object. The returned attribute can be either data or a function or method. Python has a built-in function named `hasattr` that also takes the same two parameter types and returns `True` or `False` indicating whether or not the object has an attribute with that name. Python also has a built-in function named `setattr` that takes three parameters: an object, a string, and an object to assign for the attribute. An example of this is `setattr(r, 'num', 4)`; this is equivalent to `r.num = 4`.

We have also used the terms *instance variables* and *instance methods* or just *methods* to discuss attributes of a Python class since they are the more commonly used object-oriented terminology. C++ uses the terms *instance variables* or *data members* for data and the terms *instance methods* or simply *methods* for function members. The term *members* is typically used to refer to both data members and data methods, corresponding to the Python term *attributes*.

C++ allows the interface of the class and the implementation of the class to be separated to a greater degree than Python, but does not require that they be separated. Typically the declaration of the methods and the instance variables is placed in the header file with a `.h` extension and the implementation is placed in a file with the same name except it uses a .C, .cpp, or .cc extension. We are using the .cpp extension in our examples throughout this book.

The header file defines the class name, the methods it provides, the instance variables, and sometimes the implementation of some of the short methods. The implementation file uses the `#include` preprocessor command to include the header file and provides the implementation for each of the methods (except the methods whose implementations are written in the header file). We will now examine a simplified C++ `Rational` class and cover the additional details of C++ classes starting with the header file followed by the corresponding implementation file.

```
#ifndef _RATIONAL_H
#define _RATIONAL_H

class Rational {
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
public:
  // constructor
  Rational(int n = 0, int d = 1);

  // sets to n / d
  bool set(int n, int d);

  // access functions
  int num() const;
  int den() const;

  // returns decimal equivalent
  double decimal() const;

private:
  int num_, den_; // numerator and denominator
};

#endif
```

After the `#ifndef` and `#define` preprocesser directives, the class definition starts. Note that even though this header file contains only prototypes for the methods, this is a *class definition*, not a *class declaration*. A declaration of the `Rational` class is just the code `class Rational;`. A class declaration only tells the compiler a class name exists, while a class definition specifies the name along with the instance variables and methods. Because a header file contains a class definition, the use of the `#ifndef` and `#define` processors is even more important than in a header file that just contains a number of function declarations or prototypes. Without the preprocessor directives, if the header file is included twice, you will have two definitions of the class and that is not allowed.

As in Python, the `class` keyword followed by the name of the class is used to start the class definition. C++ uses the beginning and ending braces (`{` and `}`) to mark the beginning and ending of the class definition. A semicolon is used after the ending brace for a class definition. The only places a semicolon is used after an ending brace in C++ are after class definitions, struct definitions (structs are not covered in this book), and statically initializing arrays. Forgetting the semicolon after the ending brace often leads to confusing compiler errors. Most compilers will indicate there is an error at the first line after the include statement in the file that included this header file. Many programmers immediately type the ending brace and semicolon after typing the beginning brace so they do not forget it and then enter the code between the two braces to help avoid this error. In Python, you typically specify the instance variables by initializing them in the constructor (e.g., `self.num = 0`) although other methods can create additional instance variables using the

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

same syntax. In C++, all the instance variables must be defined with their name and type inside the class definition; you cannot add new instance variables in the implementation file as you can in any Python method.

C++ supports enforced data and method protection. The keywords `public`, `private`, and `protected` are used to specify the level of protection. As you can see in the sample `Rational` class definition, the protection keyword is followed by a colon and specifies the level of protection until another protection keyword is specified. In our `Rational` example, all the methods are public and all the instance variables are private. You may specify each protection level multiple times inside a class definition if you wish, although in most cases you will only want to list each protection level once.

Any data members or methods that are `public` can be accessed by any other code; this corresponds to Python's lack of enforced protection. We discussed that the convention when writing Python code is that only the methods should be accessed by other code and that with a few exceptions such as our `ListNode` and `TreeNode` classes that are used to help implement another class, the instance variables should be accessed only by the methods of the class. Instance variables and methods that are declared `private` may be accessed only by methods of the class; the compiler will generate an error if code outside the class attempts to access a private member. Thus, in most cases instance variables should be declared `private`. There are also cases where we want some methods to be called only by other methods of the class; we saw an example of this with our `_find` method in our linked implementation of a list. The convention in Python is to name these private methods starting with one or two underscores. C++ allows you to explicitly declare methods private by listing them in a `private` or `protected` section of the class definition. This is where you declare instance variables and methods that you want to be accessed only by the methods of this class. The compiler will generate an error if code outside of the class attempts to access a private method.

The `protected` designation is similar to the `private` designation except that subclasses may also access the protected members of a class. The compiler will generate an error if any code other than the code in the class itself or a subclass attempts to access a protected method. For now, we will use only the `public` and `private` designations.

The purpose of the *constructor* in C++ is to initialize the instance variables just as it is in Python. A C++ constructor has the same name as the class and does not have a return type. As with Python, you may define a constructor that takes additional parameters, but it is a good idea to define a constructor that does not require any parameters. A constructor that does not take any parameters is

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

known as a *default constructor* whether you write it or the compiler automatically generates it. We used default parameters to allow the Rational constructor to be called with zero, one, or two parameters; because this constructor can be called without any parameters, it is a default constructor. A C++ constructor is called automatically when a variable of that type is defined (i.e., Rational r1, r2; would cause the constructor to be called for r1 and for r2). You do not need to and cannot call a constructor directly (i.e., after declaring r1 as a Rational type, you cannot write r1.Rational() or r1.Rational(2, 3)); instead you specify the parameters when you define the variable (e.g., Rational r1(2, 3);). Unlike in Python, you do not write code such as r1 = Rational() to call the constructor (you do write something similar when using dynamic memory, covered in Chapter 10); instead, you declare variables with the specified type as you do for the built-in types (e.g., int i; and Rational r;).

If you do not write any constructors, the C++ compiler implicitly creates a default constructor (it does not appear in your implementation file) with an empty body; this means the compiler does not initialize any of the instance variables. Since the compiler defined default constructor has no code, you typically want to write one to ensure that your instance variables are initialized. The default constructor is also called when you declare arrays of objects. The following variable definition causes the Rational constructor to be called 10 times, once for each item in the array: Rational r[10].

Some of the Rational methods (e.g., num(), den(), and decimal()) have the keyword const after the method declaration. This use of const indicates the method does not change any of the instance variables of the class. It should be clear that a method marked const can call only other const methods (since if it called a non-const method, that method could modify the instance variables). You may recall that we can also mark formal parameters with a const designation. For example we can write a standalone function void f(const Rational r). This means the function f is not allowed to modify the parameter so it can only call Rational methods that are designated as const methods.

Next, we will examine the syntax details of class implementation files using the Rational class as an example. To reduce space, we have left out comments, preconditions, and postconditions.

```
#include "Rational.h"

Rational::Rational(int n, int d)
{
  set(n, d);
}
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
bool Rational::set(int n, int d)
{
  if (d != 0) {
    num_ = n;
    den_ = d;
    return true;
  }
  else
    return false;
}

int Rational::num() const
{
  return num_;
}

int Rational::den() const
{
  return den_;
}

double Rational::decimal() const
{
  return num_ / double(den_);
}
```

The `Rational` implementation file includes the header file Rational.h so it has access to the prototypes for each method and the compiler can check that the proper type and number of parameters are used in the implementation. The syntax for writing methods is the return type for the method, a space, the class name, two colons, and then the method with its parameters. If the method was declared `const` in the class definition, that also must be indicated in the implementation file. Again, note that the constructor does not have a return type. The method prototypes must exactly match the return type, parameter types, and constant designations in the header file. If they do not, you will get a compiler error. Recall that we do not put the default parameter values (for the constructor in this example) in the implementation file; they appear only in the method prototype in the header file.

The two colons separating the class name and method name are known as the *scope resolution operator*. With Python, the methods are defined inside the class and the indentation indicates that the methods are part of the class. In C++ the implementation of the methods is written separately from the class definition so the class name and two colons are used to indicate that a method is part of the specified class. You may also write standalone functions that are not part of a class in a C++ class implementation file by not using the class name and the two colons. Writing

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

a standalone function in a C++ class implementation file is typically only done if the function is used by the class methods and not by any other code.

C++ does not use an explicit `self` parameter as Python does. Since the class definition specifies the names of all the instance variables, the compiler knows the names of the instance variables and does not need something similar to `self` to indicate items that are members of the class. The same is true when calling a method of the class. The methods can be called without a prefix as we called the `set` method from the constructor. C++ does contain a pointer named `this` that corresponds to Python's `self`; we will discuss it in Chapter 10 after we have discussed what a C++ pointer is.

Since an explicit indication that you are referring to instance members is not required in C++, many programmers prefix or suffix an underscore onto the names of instance variables. Use of the underscore makes it clear that you are referring to an instance variable and also allows you to use a similar name for parameters and instance variables. If a method has a formal parameter with the same name as an instance variable, the parameter makes the instance variable inaccessible unless you use the `this` pointer. If you accidently use the same name, all uses of the identifier are the parameter instead of the instance variable inside that method so your instance variables are not set or changed. The compiler does not generate an error when you name a formal parameter the same as an instance variable. This can be a difficult error to track down and is one reason many programmers add the underscore to instance variables. The explicit use of `self` in Python avoids this error. Python programmers often rely on the explicit use of `self` and name parameters and instance variables the same. Because of this, Python programmers learning C++ often make this mistake. In C++, make certain you use names for the formal parameters that are different than the class instance variables. The following example shows the problem. This example also demonstrates that you can place both the class definition and implementation code in one file; however, you do not want to do this unless your entire program is in one file. If you want to allow your program to be split among multiple files or your class to be reused in other programs, you must create a separate header and implementation file for the class.

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

class Rational {

public:
  Rational(int num_=0, int den_=1);
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
private:
  int num_, den_;
};

// this is incorrect
// do not use the same name for formal parameters and instance variables
Rational::Rational(int num_, int den_)
{
  num_ = num_;
  den_ = den_;
  cout << num_ << " / " << den_ << endl;
}

int Rational::num() const
{
  return num_;
}

int Rational::den() const
{
  return den_;
}

int main()
{
  Rational r(2, 3);

  cout << r.num() << " / " << r.den() << endl;
}
```

The output of this program on our computer is

```
2 / 3
-1881115708 / 0
```

The same problem occurs if you redeclare a local variable with the same name as an instance variable as the following example shows:

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
#include <iostream>
using namespace std;
class Rational {

public:
  Rational(int num=0, int den=1);

  int num() const;
  int den() const;

private:
  int num_, den_;
};

Rational::Rational(int num, int den)
{
  // this is incorrect
  // do not declare local variables with the same name as
  // instance variables
  int num_, den_;

  num_ = num;
  den_ = den;
  cout << num_ << " / " << den_ << endl;
}
int Rational::num() const
{
  return num_;
}
int Rational::den() const
{
  return den_;
}

int main()
{
  Rational r(2, 3);

  cout << r.num() << " / " << r.den() << endl;
}
```

The output for this example on our computer is the same as in the previous example. The instance variables are never initialized in either case so their value is whatever is in the memory location used for them before the program starts. In both cases, the actual instance variables are hidden from use in the constructor. In the first example, the formal parameters with the same name as the instance variables are the

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

variables accessed inside the constructor. In the second example the local variables are accessed in the constructor instead of the instance variables. Never use the same name for instance variables and local variables or formal parameters. The use of an underscore for instance variables (but never local variables or formal parameters) is a common technique to avoid this problem.

Another common beginner's mistake is to write code such as `r.num() = 3;` where `r` is an instance of the `Rational` class. This is not correct in Python or C++. The return value of `r.num()` is a number, not a variable in which a value can be stored. This is the same issue as incorrect code such as `4 = 3;` or `sqrt(5) = x;`. What appears on the left-hand side of the assignment statement must be a variable. The term for this is appropriately named an *l-value* since it appears on the left-hand side of the assignment statement. C++ does support a reference return type that allows a return value of a class method to be assigned a value. The details of this are covered in Chapter 10.

For functions and methods that are very short (typically less than five lines of C++ code), the overhead of making the function call takes more time than executing the actual code in the function. In these cases, it usually makes sense to avoid the overhead of a function call. C++ provides a mechanism known as *inlining* that allows you to write the code as if it is a function or method, but avoids the overhead of a function call. In effect, the compiler replaces the function call with the actual body of the function. When copying the function or method, it also properly handles the effect of passing the parameters and returning a value. For methods of a class there are two different ways to write them as inline methods. The following rewrite of our `Rational` class demonstrates both techniques. The `num()` and `den()` methods demonstrate the one technique and the `decimal` method demonstrates the other technique.

```
class Rational {

public:
  // constructor
  Rational(int n = 0, int d = 1);

  // sets to n / d
  bool set(int n, int d);
  // access functions
  int num() const { return num_; }
  int den() const { return den_; }
  // returns decimal equivalent
  double decimal() const;
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
private:
  int num_, den_; // numerator and denominator
};

inline double Rational::decimal() const
{
  return num_ / double(den_);
}
```

The `num()` and `den()` methods are written inline when they are declared. Immediately after the method definition, a semicolon is not used and instead the code follows inside braces. This technique is commonly used when the code fits on the line with the method name. The `decimal()` method is written inline after the class declaration. This is the same technique used for writing standalone functions inline that we discussed in section 8.13. The keyword `inline` is used followed by the code just as if you were writing the method in the implementation file. This technique is typically used when the code is a few lines long. The `inline` keyword is used to prevent multiple definitions of the method when multiple files include this header file. If you forget the `inline` keyword, you get a linking error indicating multiple definitions of the function if more than one file includes the header file with the method code. Inline methods should be written in the header file, not the implementation file. The exception is if the inline method is called only from one implementation file, then you could write the inline method at the top of that implementation file.

Our `Rational` constructor calls the `set` method. Notice that the method call looks like a normal function call, unlike in Python where we need to use `self` to indicate a method is being called. The reason for adding the `set` method is to prevent having two copies of code that do the same thing. It does add the overhead of an additional function call in the constructor. To solve that problem, we could make the constructor or the `set` method an inline method. It is generally a good idea to avoid duplicate code since if you change it in one place, you need to remember to change it in the other place(s) also.

With both techniques for writing a method inline in the header file, the compiler can just copy the code for the method into the function or method that called it, avoiding the overhead of a function call. Most compilers will create a normal function or method if the inline function or method is too long since copying the code for large functions will increase the size of your executable program. Whether or not the compiler actually creates an inline function is transparent to the programmer. In both cases, the return type and parameter types are checked and the parameters are effectively passed using the specified mechanism (either by value or by reference).

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

The only reason for writing inline functions is to avoid the overhead of a function call.

9.2 Strings

Now that we have learned the basics of C++ classes, we will examine the `string` class that is part of the standard C++ library. C++ strings correspond to Python's `string` data type and are used to represent sequences of characters that are usually (but not always) treated as a unit. Since C++ is for the most part backward compatible with C, it supports C-style strings and some C++ library functions require that a C string be passed as the actual parameter, so we will briefly discuss C-style strings. The C language uses an array of `char` to store string data and uses a special character `\0` to indicate the end of the string; this requires that the array size be at least one unit larger than the string of characters you want to store. Since C does not directly support classes, a C library provides separate functions that are used to manipulate the arrays of characters.

C++ strings are implemented as a class that has an array of `char` as an instance variable. As you should expect, the C++ string methods allow you to access and manipulate a string without concerning yourself with the internal implementation. The C++ `string` class provides a number of methods for manipulating the string data, but does not include all the capabilities that Python strings have. In addition to the methods the C++ `string` class supports, it also overloads many of the operators so you are able to assign and compare strings. You can read and write C++ `string` variables using the instances `cin` and `cout` and file classes defined in the `<iostream>` header file. We will not cover all the string methods, but will introduce the basics of the C++ `string` class in this section.

To use the C++ `string` class, you must `#include <string>` at the top of your file along with any other header files you are including. The `string` class is also defined within the standard namespace so you must have the statement `using namespace std` at the top of your file or refer to the class as `std::string`. When a C++ executable program reads strings using the » operator, it stops processing characters at the first whitespace (space, tab, or new line). For example, to read in a person's first and last name entered with a space between them, you would need to use two strings:

```
string first, last;
cout << "Enter your first and last name (separated by a space): ";
cin >> first >> last;
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

You may create and output strings that contain whitespace, but when using the » operator, you need to remember that it stops reading each time a whitespace character is encountered. The code `string name; name = "Dave Reed"; cout << name << endl;` works as you would expect, outputting `Dave Reed` followed by a new line. C++ provides a `getline` function that reads from the current input pointer to a delimiter; the default delimiter is the `\n` end-of-line character. The `getline` function requires two parameters: the input stream from which to read and a string that is passed by reference and will contain the string that is read. The input stream can be the `cin` instance or a file handle for reading data from a disk file. The optional third parameter for the `getline` function is the character to use as a delimiter. The `getline` function reads all the characters up to and including the delimiter and returns a string containing all the characters read except the delimiter. Using the `getline` function, we can input a first and last name as one string:

```
string name;
cout << "Enter your first and last name: ";
getline(cin, name);
```

You can mix the use of the `getline` function and the `>>` operator with `cin` or a file handle, but it requires that you carefully process the input. When you use `cin` to read a variable, it skips leading whitespace, but leaves the trailing whitespace, including the new line character in the input stream. The `getline` function reads everything up to the delimiter, including the delimiter, so if a `getline` follows a `cin` that reads everything on the line, it gets an empty string. You must make two calls to `getline` in this case, and the second one will get the data on the next line.

The C++ `string` class supports the standard comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=`. The rules for comparison are the same as in Python; dictionary order is used and lowercase letters are greater than uppercase letters since the ASCII codes for lowercase letters are larger. Unlike Python strings, C++ strings are mutable. You can both access individual characters and set individual characters using the brackets operator (`[]`). As you should expect, the indexing starts at zero and you cannot use negative values since internally the string is represented as a C++ array. There is no range checking, so you need to ensure that you do not access beyond the end of the string. C++ strings also support the assignment operator `=` for assigning a string variable or expression on the right-hand side of the assignment statement to the string variable on the left-hand side.

The C++ string assignment operator creates a separate copy of the data, unlike Python which would have two references to the same data. If after assigning one C++ string variable to another, you change one of the strings, it does not change the

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

other. The + and += operators work the same as they do in Python. The following example demonstrates some of these concepts.

```
// stringex.cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string first = "Dave";
  string last = "Reed";
  string name;

  name = first + " " + last;
  cout << name << endl;

  first[3] = 'i';
  first += "d";

  name = first + " " + last;
  cout << name << endl;
  cout << name.substr(6, 4) << endl;
  return 0;
}
```

The preceding example outputs `Dave Reed`, `David Reed`, and `Reed` on three separate lines. Notice that the single quotation mark is used with the bracket operator since `first[3]` is a single character. You cannot use Python's slicing syntax for accessing a substring; C++ does provide a `substr` method. Its prototype is `string substr(int position, int length)`. It returns a string starting at the specified starting position with the specified length. This is different than Python slicing which takes the starting and ending positions. The `string` class also has a method named `c_str()` for returning a C array of characters. This is useful when you need to call a function that requires a C-style string instead of a C++ string. The `find` method takes a string to search for and optional starting position for the search. It returns the index of the first occurrence of the search string in the string. There are a number of additional `string` methods, but these are a few of the ones that are commonly used.

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

9.3 File Input and Output

File input and output often involves the use of strings although you can input ASCII numeric data directly as numbers or read a file in a binary format corresponding directly to how the computer represents an internal data type. We will not cover the reading of binary files in this book. C++ uses instances of classes to perform file input and output as it does for keyboard and monitor input and output. The `fstream` header file contains the class declarations of `ifstream` and `ofstream` for file input and output, respectively. These are also in the namespace `std`. Similarly as in Python, you must associate the file variable with a filename using the `open` method. The following example demonstrates file input and output in C++ by prompting the user for a file name and writing the string `David Reed` to the file. It then opens the file for reading, reads the first line in the file using the `getline` function, and outputs it using the `cout` statement.

```
// getline.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  string filename, name, first, last;
  ofstream outfile;
  ifstream infile;

  cout << "Enter file name: ";
  cin >> filename;
  outfile.open(filename.c_str());
  outfile << "David" << " " << "Reed" << endl;
  outfile.close();
  infile.open(filename.c_str());
  getline(infile, name);
  cout << name << endl;
  infile.close();
  return 0;
}
```

Notice that the `open` method requires the C version of a string which is an array of characters, so we need to use the `c_str()` method of the `string` class when opening the file. As with Python, you need to close the file to ensure that data written to the file is flushed to the disk. In this example we demonstrated the use of the `getline` function although we could have followed the same pattern as we did

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

when writing the file and read two separate strings and combine them using the + operator. The code fragment for this method is

```
infile.open(filename.c_str());
infile >> first >> last;
infile.close()
name = first + " " + last;
cout << name << endl;
infile.close();
```

You can also read numeric data from an ASCII file using a similar technique. You open the file and then specify a numeric data variable (`int`, `float`, or `double`). Just as when reading numeric values using the keyboard, whitespace (space, tab, or new line) is used to separate numeric values and the amount of whitespace does not matter. Each time you attempt to read a value, it skips past any whitespace to attempt to find a numeric value. If it encounters any non-numeric characters immediately after any preceding whitespace while attempting to read a number, a run-time error is generated. When reading a number with a non-numeric digit after it, it reads the number, but not the other non-numeric digit, leaving the file pointer at that location. The next input will start with that character. The following would read a file named `in.txt` containing 10 integer values as ASCII text with each one separated by any amount of whitespace and output each value on a line as it reads it.

```
// readfile.cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  ifstream ifs;
  int i, x;
  ifs.open("in.txt");
  for (i=0; i<10; i++) {
    ifs >> x;
    cout << x << endl;
  }
  return 0;
}
```

The `open` method of both the `ifstream` and `ofstream` classes has a second parameter for specifying the mode for opening the file. It should be clear from the preceding examples that the second parameter has a default value. This book does

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

not cover the details of the second parameter or how to read or write binary files with C++.

9.4 Operator Overloading

As you may have determined based on the discussion of strings, C++ supports user-defined operator overloading. As with Python, the purpose of operator overloading is to allow for more concise, readable code. Because C++ does not use references by default, it is also necessary to use operator overloading to override the assignment operator for classes that use dynamic memory; we will discuss this in Chapter 10.

With C++, you may choose to make the operators methods of the class or standalone functions (a few must be standalone functions). Some programmers prefer the standalone functions, since the binary operator functions take two parameters corresponding to the two instances of the class to which the operator is applied. If you implement the operator as a method of the class, only one parameter appears in the method prototype; the left parameter for the operator is the implicit parameter corresponding to the instance with which the method was called. In Python both parameters appear in the definition since the `self` parameter is explicit. The drawback of using standalone functions is they cannot access the private data of the class. Because of this, the class must provide methods to access and possibly modify the private data. C++ also provides a `friend` construct for allowing certain functions or methods from other classes to access the private data. We will examine this technique when we learn how to overload the input and output operators.

C++ names the methods for operator overloading using the word `operator` followed by the actual symbol for the operator that is being overloaded. We will first examine the technique where the operator is not a member of the class, so we will be writing standalone functions. The following is the complete `Rational` header and implementation file for the addition operator written as a standalone function.

```
// Rationalv1.h
class Rational {

public:
  // constructor
  Rational(int n = 0, int d = 1) { set(n, d); }
  // sets to n / d
  bool set(int n, int d);

  // access functions
  int num() const { return num_; }
  int den() const { return den_; }
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
  // returns decimal equivalent
  double decimal() const { return num_ / double(den_); }

private:
  int num_, den_; // numerator and denominator
};

// prototype for operator+ standalone function
Rational operator+(const Rational &r1, const Rational &r2);
```

```
// Rationalv1.cpp
#include "Rationalv1.h"

bool Rational::set(int n, int d)
{
  if (d != 0) {
    num_ = n;
    den_ = d;
    return true;
  }
  else
    return false;
}

Rational operator+(const Rational &r1, const Rational &r2)
{
  int num, den;

  num = r1.num() * r2.den() + r2.num() * r1.den();
  den = r1.den() * r2.den();
  return Rational(num, den);
}
```

Note that since the operator is a standalone function, the class name and two colons (`Rational::`) is not placed in front of the name of the function (`operator+`). A sample program that calls the operator is

```
// mainv1.cpp
#include "Rationalv1.h"
int main()
{
  Rational r1(2, 3), r2(3, 4), r3;

  r3 = r1 + r2; // common method of calling the operator function
  r3 = operator+(r1, r2); // direct method of calling the function
}
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Since the function is not a member of the class, it cannot access the private data members directly and needs to use the public methods to access the numerator and denominator. The function prototypes for the standalone function version of many of the operators that can be written are summarized in the following table (this is not a complete list). For other classes, you obviously need to replace `Rational` with the name of that class type. We pass the parameters as `const` reference parameters; this means only `const` methods of the `Rational` class can be called inside these functions. This does not cause a problem since applying any of the operators should not change the parameter(s). Remember that the reason for passing class instances using the `const` designation and by reference is that when using pass by reference, only the address of the object is passed. This results in less data being copied than if we used pass by value so it is faster and uses less memory. The first column in the table shows the prototype for the function. The second column shows how the function/operator is called for two instances of the `Rational` class and what result it computes and returns.

Function	Computes
`Rational operator+(const Rational& r1, const Rational& r2)`	`r1 + r2`
`Rational operator-(const Rational& r1, const Rational& r2)`	`r1 - r2`
`Rational operator*(const Rational& r1, const Rational& r2)`	`r1 * r2`
`Rational operator/(const Rational& r1, const Rational& r2)`	`r1 / r2`
`Rational operator-(const Rational& r1)`	`-r1`
`bool operator<(const Rational& r1, const Rational& r2)`	`r1 < r2`
`bool operator<=(const Rational& r1, const Rational& r2)`	`r1 <= r2`
`bool operator>(const Rational& r1, const Rational& r2)`	`r1 > r2`
`bool operator>=(const Rational& r1, const Rational& r2)`	`r1 >= r2`
`bool operator==(const Rational& r1, const Rational& r2)`	`r1 == r2`
`bool operator!=(const Rational& r1, const Rational& r2)`	`r1 != r2`

The operator overloading code can also be written as a method (i.e., a member of the class). Usually the operator would be written in the .cpp file and the prototype for it would be written in the .h file. Since we are writing a member method, the prototype needs to be declared in the `public` section of the class declaration. The object the method is called with is the implicit first parameter `r1` that is visible in the function version, so it is not used in the method version. The following shows the header file and implementation file for the addition operator written as a method of the class.

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
// Rationalv2.h
class Rational {

public:
  // constructor
  Rational(int n = 0, int d = 1) { set(n, d); }

  // sets to n / d
  bool set(int n, int d);

  // access functions
  int num() const { return num_; }
  int den() const { return den_; }

  // returns decimal equivalent
  double decimal() const { return num_ / double(den_); }

  Rational operator+(const Rational &r2) const;

private:
  int num_, den_; // numerator and denominator
};
```

```
// Rationalv2.cpp
#include "Rationalv2.h"

// code for set method is also required
// see previous example for the code

Rational Rational::operator+(const Rational &r2) const
{
  Rational r;

  r.num_ = num_ * r2.den_ + den() * r2.num();
  r.den_ = den_ * r2.den_;
  return r;
}
```

Since the method is a member of the class, it can directly access the private data members of any instance of the class. Also note that the first parameter is implicit in the method prototype as it is in all C++ class methods. Because of this, that instance's data and methods are accessed by specifying the name of the data/method member without a variable name before it while the explicit second parameter's (`r2`) data is accessed by specifying the name of that parameter followed by a period and then the data/method member. The preceding example uses both `num_` and `den()` to demonstrate that instance variables and methods, respectively,

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

can be accessed directly for the implicit parameter; normally you would pick one style and use it consistently. Some programmers prefer the non-member function so that the function prototype is symmetric and shows both parameters. Others prefer the class method so all the code is encapsulated within the class and the methods can access the private data.

The common way of calling the method is using the operator notation as we did when using the function technique for writing operators. The direct way of calling it is the standard syntax for calling a method (i.e., a class instance, followed by a period, followed by the method name).

```
// mainv2.cpp
#include "Rationalv2.h"
int main()
{
  Rational r1(2, 3), r2(3, 4), r3;

  r3 = r1 + r2; // common method of calling the operator method
  r3 = r1.operator+(r2); // direct method of calling the operator
}
```

The following table shows the prototypes for the operators when they are members of the class. The second column again shows how to call the methods and what value the operator computes and returns.

Method	Computes
Rational operator+(const Rational& r2)	r1 + r2
Rational operator-(const Rational& r2)	r1 - r2
Rational operator*(const Rational& r2)	r1 * r2
Rational operator/(const Rational& r2)	r1 / r2
Rational operator-()	-r1
bool operator<(const Rational& r2)	r1 < r2
bool operator<=(const Rational& r2)	r1 <= r2
bool operator>(const Rational& r2)	r1 > r2
bool operator>=(const Rational& r2)	r1 >= r2
bool operator==(const Rational& r2)	r1 == r2
bool operator!=(const Rational& r2)	r1 != r2

If you wish to override the input (») and output («) operators, they must be written as standalone functions. The reason for this is the first parameter of a method must be an instance of that class. Consider the code `cin » r1`. You might be tempted to write it as a member method, but recall that this would imply the method would be called as `cin.operator»(r1)`. Since `cin` is not an instance of

04/25/2018 - RS00000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

the `Rational` class, the input operator cannot be a member of the `Rational` class and must be written as a standalone function. This is also the case when using the output operator « with an instance of the `ostream` class such as `cout`. A standalone function of the output operator for our `Rational` class is

```
std::ostream& operator<<(std::ostream &os, const Rational &r)
{
  os << r.num() << "/" << r.den();
  return os;
}
```

The operator needs to return the instance of the output stream variable `os` that is of type `ostream` so that it can be chained together (e.g., `cout « r1 « r2`). In this example, the returned result of `cout « r1` needs to be the `ostream` instance `cout` so it is now the first parameter to the call for outputting `r2`. The `ostream` parameter `os` also needs to be passed by reference and returned as a reference since outputting the variable to the stream changes the stream. We will cover returning by reference in more detail in Chapter 10, but for now just learn the syntax for returning by reference which is appending an ampersand onto the return type (e.g., `ostream&` for the output operator).

Since the operator is a non-member function, it cannot access the private data of the `Rational` class. There are times where we want to allow certain other classes or certain functions to be able to access the private data of a class. C++ provides a mechanism for permitting this using the `friend` keyword. One common example where allowing a non-member function to access the private members directly makes sense is the input/output operator functions. Another example would be our `ListNode` class. We may want to allow the `LList` class to access the `ListNode` data members directly since those two classes are tightly coupled together. A function or class is specified as a `friend` inside the class that wants to make it a friend. The following code example demonstrates this for our `Rational` class. If we wanted to make an entire class a friend, an example of the syntax is `friend class LList`. If we placed that line inside our `ListNode` class then all the `LList` methods would have access to the private data of the `ListNode`. We will demonstrate a complete example of this when we examine linked structures using C++ in Chapter 11.

The following code is the header file for the complete, simplified `Rational` class demonstrating operator overloading and friends. For brevity, the pre- and postconditions and comments are not included for all the methods.

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
// Rationalv3.h
#ifndef _RATIONAL_H
#define _RATIONAL_H

// needed for definition of ostream and istream classes
#include <iostream>

class Rational {

// declare input and output operators functions as friends
// to the class so they can directly access the private data
friend std::istream& operator>>(std::istream& is, Rational &r);
friend std::ostream& operator<<(std::ostream& os, const Rational &r);

public:
  // constructor
  Rational(const int n = 0, const int d = 1) { set(n, d); }

  // sets to n / d
  bool set(const int n, const int d);

  // access functions
  int num() const { return num_; }
  int den() const { return den_; }

  // returns decimal equivalent
  double decimal() const;

private:
  int num_, den_; // numerator and denominator
};

// prototypes for operator overloading
Rational operator+(const Rational &r1, const Rational &r2);

// declare the non-member input output operator functions
std::istream& operator>>(std::istream &is, Rational &r);
std::ostream& operator<<(std::ostream &os, const Rational &r);

#endif
```

The corresponding .cpp implementation file is

```
// Rationalv3.cpp
using namespace std;
#include "Rationalv3.h"

```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
bool Rational::set(const int n, const int d)
{
  if (d != 0) {
    num_ = n;
    den_ = d;
    return true;
  }
  else
    return false;
}

Rational operator+(const Rational &r1, const Rational &r2)
{
  int num, den;

  num = r1.num() * r2.den() + r2.num() * r1.den();
  den = r1.den() * r2.den();
  return Rational(num, den);
}

std::istream& operator>>(std::istream &is, Rational &r)
{
  char c;

  is >> r.num_ >> c >> r.den_;
  return is;
}

std::ostream& operator<<(std::ostream &os, const Rational &r)
{
  os << r.num() << "/" << r.den();
  return os;
}
```

The `Rational` object passed to the input operator function must be passed by reference since we want the value we read to be stored in the actual parameter sent (i.e., when we execute `cin » r` we want the value the user enters to be stored in `r`). This is also why it cannot be passed as a `const` parameter. To allow us to type in a value such as `2/3`, we need to read the forward slash in the input operator function. We declare the variable `c` as a `char` to store the slash but ignore the value after reading it since our `Rational` class encapsulates the number by storing two integers.

Also notice in our example that we did not put the `using namespace std` line in the header file. Instead we used the prefix syntax `std::` when referring to the names of the `ostream` and `istream` classes that are defined within the `std` namespace. Remember that the reason for this is that if we had put the `using namespace std`

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

line in the header file, any file that included our Rational.h file would effectively have the `using namespace std` line in it. For this reason, you should never put a `using` statement in a header file. We did put the `using namespace std` line in our Rational.cpp file so that we did not need to write `std::` in front of all the names defined in the namespace; this is not a problem since you never include an implementation (.cpp) file.

9.5 Class Variables and Methods

C++ also supports a mechanism for creating class variables. You may recall that we discussed how to create class variables in Python in subsection 2.3.2. With instance variables, each instance of a class gets its own separate copy of the instance variables. With class variables, all instances of the class share the same variable (i.e., there is only one copy of the class variable no matter how many instances of the class exist). The `Card` class we discussed in subsection 2.3.2 is a good example in which using class variables makes sense. We will create a similar `Card` class in this section using C++ class variables.

```
// Card.h
#ifndef __CARD_H__
#define __CARD_H__
#include <string>

class Card {
public:
  Card(int num=0) { number_ = num; }
  void set(int num) { number_ = num; }
  std::string suit() const;
  std::string face() const;
private:
  int number_;
  static const std::string suits_[4];
  static const std::string faces_[13];
};
inline std::string Card::suit() const
{
  return suits_[number_ / 13];
}
inline std::string Card::face() const
{
  return faces_[number_ % 13];
}
#endif // __CARD_H__
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

The mechanism for creating class variables is to declare them with the `static` prefix. In C++, there are a number of different uses for the keyword `static` and it is easy to confuse them. This use of `static` has a completely different meaning than the use of `static` we discussed in the previous chapter to create local variables that always use the same memory location. Declaring an instance variable `static` indicates it is a class variable and thus there is only one copy of that variable that all instances of the class share. Using a class variable in our example makes sense since we do not need a separate copy of the face and suit names for each instance of the class. Making these instance variables would be a huge waste of memory. With class variables, each instance of our class requires only four bytes of memory. If the face and suit name variables were not class variables, each instance of our `Card` class would require around 100 bytes to store the number and all the strings. The following is the implementation file for the `Card` class.

```
// Card.cpp
#include "Card.h"

const std::string Card::suits_[4] = {
  "Hearts", "Diamonds", "Clubs", "Spades" };

const std::string Card::faces_[13] = {
  "Ace", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine",
  "Ten", "Jack", "Queen", "King" };
```

Class variables are defined as if they were non-local variables (i.e., outside of any function) and the variables are initialized using the assignment statement once when the program is first executed. Since there is only one copy of the class variables, we do not want to assign the values inside the constructor. We declared the class variables with the `const` prefix in the header file so once we initialize the variables with these statements, we cannot change their values. Even if the class variables were not declared with the `const` prefix, we would still need to define them once in the implementation file (with or without providing initial values). A class definition does not actually cause any memory to be allocated; it is only when we create an instance of the class that memory is allocated. This is why we must define the class variables in an implementation file so that memory is allocated for them.

The following is a sample program that uses our `Card` class containing the class variables.

```
// test_Card.cpp
#include <iostream>
using namespace std;
#include "Card.h"
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
int main()
{
  Card c[52];
  int i;

  for (i=0; i<52; ++i) {
    c[i].set(i);
  }
  for (i=0; i<52; ++i) {
    cout << c[i].face() << " of " << c[i].suit() << endl;
  }
  return 0;
}
```

Even though there is no need to do this, what would happen if we tried to put the statement `cout « Card::faces_[0] « endl;` in our `main` function? This does demonstrate the correct usage of accessing a class variable using the class name followed by two colons followed by the name of the class variable. However, the class variables were declared `private` so they are not accessible outside of the class even though the variable definitions are not inside the class. If we declared the class variables in the `public` section this would work.

You may be wondering why we needed to create a separate implementation file since all the methods were defined inline in the header file. If we instead put the class variable definitions in the header file as the following code shows, we could end up with the same names being defined multiple times. Recall that each variable or function can have only one definition.

```
// this code should not be used

#ifndef __CARD_H__
#define __CARD_H__

#include <string>

class Card {
public:
  Card(int num=0) { number_ = num; }
  void set(int num) { number_ = num; }
  std::string suit() const;
  std::string face() const;
private:
  int number_;
  static const std::string suits_[4];
  static const std::string faces_[13];
};
```

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
const std::string Card::suits_[4] = {
  "Hearts", "Diamonds", "Clubs", "Spades" };

const std::string Card::faces_[13] = {
  "Ace", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine",
  "Ten", "Jack", "Queen", "King" };

//--------------------------------------------------------------------

inline std::string Card::suit() const
{
  return suits_[number_ / 13];
}

inline std::string Card::face() const
{
  return faces_[number_ % 13];
}

//--------------------------------------------------------------------

#endif // __CARD_H__
```

This header file works correctly if only one file includes it since that creates one definition of the class variables `suits_` and `faces_`. However, if multiple implementation files that are used to create one executable program include this header file, then we have multiple definitions of the class variables and we get a linker error indicating multiple definitions of the symbols. For this reason, class variables should always be defined in an implementation file as our original example did.

In our example the class variables were declared `const` since it does not make sense to change them. But in some cases you may want class variables that are not `const`. One possible use of a non-`const` class variable is to keep track of the number of instances of the class that are created. To do this, we create a class that has the constructor increment the class variable. The value of this class variable tells us the total number of instances of the class that have been created. To do this, we add a class variable to the class using the following line inside the class definition in the header file: `static int count_;`. We then add the line `int Card::count_ = 0;` to the implementation file. If we declare the class variable in the `public` section of the header file, then we can access it directly. This would allow us to put the following line in our `main` function: `cout « Card::count_ « endl;`. Of course, normally you do not want to declare data members of a class in the `public` section. Someone could put the line `Card::count_ = 100;` in their code and destroy the

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

integrity of the value `count_` storing the number of instances of the `Card` class that have been created.

Classes can also have class methods that are called without an instance of the class. Using a class method to access the class variable `count_` is the proper way to ensure the integrity of the data. We need to add a class method that returns the value of the class variable. Class methods are also declared with the `static` prefix. The declaration and definition of the method is `static int count() { return count_; }`. We call the method using the code `cout « Card::count() << endl;`. You should realize that class methods can access class variables, but they cannot access instance variables. The reason for this is that when calling a class method, you are not specifying an instance of the class as we do when we call an instance method (e.g., `Card::count()` vs. `c.face()`). A class method cannot know which instance data to use since an instance is not specified when the method is called.

You may have noticed that our sample code to count the number of cards never decreases the class variable storing the number. This means the class variable will store the number of instances that have been created even though some of them may not exist. To make the class variable indicate the number of instances of the class that currently exist as the program is executing, we need to decrease the value of the class variable when the lifetime of a `Card` instance ends. We will learn in Chapter 10 about destructors; they could be used to accomplish this task.

9.6 Chapter Summary

This chapters covers the syntax and concepts for writing and using C++ classes. The following is a summary of some of the important concepts.

- C++ classes are usually written in two parts that are in separate files: the class definition in a header file and the code for the methods in an implementation file.
- A semicolon must be placed after the ending brace of a class definition.
- C++ constructors have the same name as the class and are called automatically when a variable of that type is defined.
- Programmers commonly prefix or suffix an underscore onto instance variables so they do not accidently use the same identifier name for instance variables as they do for formal parameters and local variables.

04/25/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++