Appendix

L

C++ for Python Programmers

Contents

This appendix covers some of the differences between Python and C++ that a beginning programmer may encounter. While there are many additional differences, they either are not applicable to the code and examples used in this text or are explained when needed. This appendix should be used in conjunction with Appendix A to fully understand the features provided in C++.

L.1 General Concepts

Transitioning from the Python programming language to C++ is similar to graduating from high school and going through basic training in the military. Python is a weakly typed, interpreted language with a readable syntax. C++, on the other hand, is a strongly typed, compiled language that follows strict punctuation rules for its syntax. The good news is that after completing boot camp, you have the chance to become an Army Ranger or Navy Seal. This text can help you be all you can as a computer scientist and C++ programmer.

Python programs are interpreted and can execute on any computer that has a Python interpreter. Python interpreters have even been written in JavaScript, so that you can run Python code inside a window in your web browser! C++ programs, on the other hand, are compiled into object code that is specific to an operating system. This means that you need to recompile your C++ code for each operating system that you want your program to run on. The code in this book follows the recent ANSII C11 Standard, so you should be able to compile and run your programs on whichever operating system you choose if your compiler supports that standard.

L.2 Comparison of Language Elements

When it comes to writing a C++ program, there are several important differences you need to keep in mind.

Comments. Comments are an important part of any program, and C++ supports single and multi-line comments with different delimiters:

```
// This is a comment that appears on its own line.
int  sum = 0; // This is a comment that appears within a C++ statement.
// The following comment begins with /** and ends with */.
/** This is a special multiline comment used for documentation systems
    such as doxygen. See Appendix I for details. */
// The following comment begins with /* and ends with */.
/* This is a multiline comment that is often used to temporarily
    disable a group of statements during debugging. */
```

Statements. White space and indentation in C++ have no significance. C++ statements end with a semicolon. Statements in a block, known as a compound statement, are analogous to a suite in Python and are grouped together by braces `{ }`.

Variables. In Python, a variable can be used as soon as it is assigned a value. Variables in C++ must be declared and given a type before they can be used. Though the C++ compiler does not require it, variables should also be given an initial value before being used. Once a variable has been declared with a given a type, it maintains that type throughout the scope in which it was declared.

Relational operators. Variables in C++ can be used only with one relational operator at a time. Python expressions such as `5 < x < 10` must be written as `(5 < x) && (x < 10)` in C++. The operator `&&` is used for a logical AND, and `||` is used for a logical OR.

Predicate expressions. Predicate expressions in conditional and iteration flow-control statements must be in parentheses.

Assignment statements. Multiple assignment is not allowed in C++. Python statements such as `x,y = 5,6` cannot be used.

Iteration statements. C++ has a `while` statement that behaves like the Python `while` statement. The C++ `for` statement is a special case of the `while` statement. To use a C++ `for` statement as you would use a Python `form` statement, you must use iterators, which are explained in C++ Interlude 6. C++ has a `do-while` statement, which is a post-test loop. Python has no equivalent statement.

Basic input and output. In a Python application, you use the `raw_input` function to accept input from the keyboard. This function returns a string. If you need an integer or a value of some other data type, you must use another function to transform the string into the desired input. Output to the display is accomplished by calling the `print` function. For example, the following Python code displays a prompt for the user, accepts a response as an integer, and echoes the response back to the user:

```
userNumber = int (raw_input ("How many apples are in a box? "))
print "You entered %d apples per box."%( userNumber)
```

C++ uses two objects, `cin` and `cout`, for keyboard input and console display output, respectively. To use `cin` and `cout`, you should include the following statements at the start of your program:

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

The following C++ statements that are equivalent to the previous Python statements:

```
cout << "How many apples are in a box? ";
int  applesPerBox = 0;
cin >> applesPerBox;
cout << "You entered " << applesPerBox << " apples per box." << endl;
```

Notice the use of the output stream operator << to send string literals and variables to the output object `cout`. Similarly, the input stream operator >> sends input from the input object `cin` to a variable. It is easy to remember which operator to use if you imagine them as arrows pointing in the direction of the data flow.

The constant `endl` is defined in `iostream` and represents the new-line character. Additional information about the `iostream` library and basic input and output in C++ is in Section A.2 of Appendix A.

Example. Listing L-1 gives a short interactive Python program using the statements and expressions just discussed. Listing L-2 shows the equivalent program in C++. C++ programs begin execution at the `main` function. You can see some of the differences in user input and output in this example also.

LISTING L-1 Short Python program to test and modify a user-entered value

```
small, large = 1, 1000
print  "Your number was between %d and %d large." %(small, large)
userNumber = int (raw_input ("Enter something: "))

if  small < userNumber < large :
    print  "Your number was between %d and %d large."%(small, large)

elif  userNumber < small :
    while  userNumber <= small :
        userNumber = userNumber + 1

else :
    while True  :
        userNumber = userNumber - 1
        if  userNumber < 1000 :
            break
        print  "I fixed your number to be between %d and %d!" %(small, large)
```

LISTING L-2 Short C++ program equivalent to the Python program in Listing L-1.

```
#include <iostream>
using namespace  std;
int  main()
{
   int  small = 1;
   int  large = 1000;
   int  userNumber = 0;
   cout << "Enter something: ";
   cin >> userNumber;
```

(*continues*)

```
    if ( (small < userNumber) && (userNumber < large) )
    {
        cout << "Your number was between " << small << " and ";
        cout << large << endl;
    }
    else if (userNumber < 1)
    {
       while (userNumber <= 1)
       {
          userNumber++;
       }
    }
    else
    {
       do
       {
          userNumber--;
       } while  (userNumber >= 1000);
       cout << "I fixed your number to be between " << small << " and ";
       cout << large << endl;
    }
}
```

L.3 Functions

As with its variables, C++ functions are strongly typed. When a C++ function is defined, the parameters are given a type. If the function returns a value, the function must indicate the type of the return value. This information is given in the header, that is, the first line, of the function. If a function does not return a value, its return type is `void`. The definition of the function follows the header and is enclosed in `{ }`, as in other C++ blocks.

Here is a short Python function that computes the cube of its single parameter:

```
def  realCube(x) :
     return  x * x * x
```

Here is the equivalent C++ function:

```
double realCube(double  x)
{
   return  x * x * x;
}
```

Notice that the function and its parameter are each given a type. C++ functions are discussed in greater detail in Section A.2 of Appendix A.

Any function defined after `realCube` in the file can invoke `realCube`. But what if a function is defined before `realCube` and needs to call it? In this case, the compiler would issue an error stating that `realCube` was not defined. C++ requires that each function, constant, variable, or class be defined before it can be called.

In a simple program with a few functions, it is easy to order the function definitions so that the compiler sees each one before it encounters the function call. In more complex programs, this is more difficult, or even impossible, to achieve. However, the compiler needs only the header of a function before processing a call to it, and C++ provides a way for you to satisfy the compiler. You simply write

the function's header followed by a semicolon—that is, its prototype—at the beginning of the program that calls it. For example, the prototype of the function `realCube` is

```
double realCube(double  x);
```

Then you can place the function's definition anywhere within your program.

In larger programs, function prototypes are placed into one or more header files, as described in Section K.1 of Appendix K. You then include the header files in your program. Prototypes are also important in the creation of classes, as you will see in the next section. Section A.8.1 of Appendix A also discusses header files.

Note that you can pass a function as an argument to another function or method. This is discussed in Section 15.2.1 of Chapter 15.

L.4 Classes

Python classes permit clients of the classes to access all data fields and methods. In C++, this is called **public access** and is recommended only for methods. In Python, when you create a class member that should be used only by class methods, you typically begin the member's name with two underscores, __. This convention is only a signal to the programmer; the Python interpreter does not enforce it. C++ permits the class designer to restrict access to certain data fields and methods by labeling them as **private**, and the compiler enforces this restriction by permitting only class methods to access private members.

Classes in C++ have constructors that are similar to the `__init__` method in a Python class. When writing a method in a Python class, you precede the names of data fields by `self` to indicate that they are instance variables of the class. Instance variables in a C++ class do not need a special prefix, if they are referenced from within a method of the class.

Here is a simple Python class, `PlainBox`:

```
class PlainBox

     def __init__(self, theItem):
         self.item = theItem

     def setItem(self, theItem):
         self.item = theItem

     def getItem(self) :
         return self.item
```

You also can define the same class in a form that is closer to that of a C++ class:

```
class PlainBox

     __item = " "

   def __init__(self, theItem):
         self.__item = theItem

   def setItem(self, theItem):
         self.__item = theItem

   def getItem(self) :
         return self.__item
```

Though is it possible to define a class in C++ in a single file, we will use two files, since this is a more common and flexible approach. The first file is the class header file; it contains the data fields for the class and prototypes for all methods defined in the class. It is a description of the class and does

not normally contain executable code. For example, here is the header file for the C++ version of the previous Python class:

```
/** @ file PlainBox.h */
#include <string>    // Needed because we use string objects.
using namespace std; // Strings are part of the system std library.

class PlainBox       // This is a class declaration for PlainBox.
{
   private:          // Items defined below have private access.
      string item;

   public:           // Items defined below have public access.
      // Prototypes for constructors and methods:
      PlainBox();
      PlainBox(string theItem);
      void  setItem(string theItem);
      string getItem();
}; // end PlainBox - Note semi-colon after declaration.
```

Notice that each data member and method does not need to have an access modifier. Private data members and methods are grouped together, and public methods are grouped with each other. A colon follows each access modifier. It is possible to have more than one private, protected, or public section, but it is not common. The class declaration must end with a semi-colon after the closing brace, as shown above.

The **implementation file**, or **source file**, for the class typically has the same name as the header file, but ends in `.cpp` to indicate that it is a C++ file. The implementation file contains the definition for the methods declared in the accompanying header file. Here is the implementation file for the C++ class `PlainBox`:

```
/** @ file PlainBox.cpp */

#include "PlainBox.h" // We need to tell the compiler about our data
                      // fields and methods.

PlainBox::PlainBox()
{
}  // end default constructor

PlainBox::PlainBox(string theItem)
{
   item = theItem;
}  // end constructor

void  PlainBox::setItem(string theItem)
{
  item = theItem;
}  // end setItem

string PlainBox::getItem()
{
   return  item;
}  // end getItem
```

This implementation file begins by including the header file, so that the compiler has the class declaration and knows the data fields and methods of this class. We then define each method. In the implementation file there is no "master" set of braces enclosing all of the methods. Each method is individually defined. To let the compiler know that a method belongs to the `PlainBox` class, we add

the prefix `PlainBox::` to the method name. Observe that the prefix comes after the method's return type. The prefix is an extension of the method name only.

To create a `PlainBox` object in Python, you could use the following statement:

```
myBox = PlainBox("Jewelry");
```

C++ provides several forms for creating objects:

```
PlainBox myBox = PlainBox("Jewelry");
PlainBox myBox("Jewelry");
```

These statements are equivalent in C++ and create a `PlainBox` object that is local to the function or method in which it is created. In these examples, the C++ keyword `new` is not used.

The following example uses the keyword `new` to create a `PlainBox` object that exists until the programmer specifically deletes it:

```
PlainBox* someBoxPtr = new  PlainBox("Jewelry");
```

Notice that `PlainBox` is followed by the `*` character to indicate that `someBoxPtr` is a pointer to a `PlainBox` object and is not the object itself. Pointers are discussed in more detail in C++ Interlude 2.

If an object in C++ is instantiated using the `new` operator, as in the example above, it is the responsibility of the programmer to delete that object from memory when it is no longer needed. In Python, garbage collection deletes unused objects and frees the memory that was used by those objects. C++ does not provide garbage collection, so you must free unused memory. To delete the object pointed to by `someBoxPtr`, use the statements

```
delete  someBox;     // Frees memory used by the object.
someBox = nullptr ; // A safety precaution.
```

Each time you use the `new` operator to create an object, you must write a corresponding `delete` statement to free the object's memory. Otherwise, your application will have a memory leak.

Additional information about C++ classes appears in Section A.8 of Appendix A, Chapter 1, and in C++ Interludes 1 and 4.