A Transition Guide from Python 2.x to C++

Michael H. Goldwasser David Letscher

Saint Louis University

August 2011 revision

This is a supplement to the book
Object-Oriented Programming in Python,
Prentice-Hall, 2007
ISBN-13: 978-0136150312.

Contents

1 Introduction

Python is a wonderful programming language. However, more than a thousand programming languages have been developed over time, with perhaps a hundred that are still actively used for program development. Each language has its own strengths and weaknesses in trying to support the development of efficient, maintainable, and reusable software. Software professionals must become accustomed to programming in different languages. Fortunately, once a person has a solid foundation in one language, it becomes easier to transition to programming in another language.

This document is designed for Python programmers choosing C++ as a second language. C++ is a widely used language in industry and, as an object-oriented language, it has much in common with Python. Yet there exist significant differences between the two languages. This transition guide is not meant to serve as a complete, self-contained reference for C++. Our goal is to provide an initial bridge, built upon the knowledge and terminology introduced in our book *Object-Oriented Programming in Python*.

C++ is a direct extension of an earlier programming language named C. The C programming language was introduced in 1973 and widely used for software development for decades (in fact, its use is still prevalent). Its greatest strength is its run-time efficiency, however it is not object-oriented. In the early 1980s, developers at Bell Labs began work on C++, adding support for object orientation while preserving aspects of the original syntax of C. As a result, C++ provides a more robust set of existing libraries while still providing the ability to create fast executables.

C and C++ provide great flexibility in controlling many of the underlying mechanisms used by an executing program. A programmer can control low-level aspects of how data is stored, how information is passed, and how memory is managed. When used properly, this control can lead to a more streamlined result. Furthermore, because of the long history of C and C++ and their widespread use, the technology has been highly optimized.

The greatest weakness of C++ is its complexity. Ironically, this weakness goes hand-in-hand with the very issues that we described as strengths of the language. With decades of prominence, its evolution has been somewhat restricted by the desire to remain backward compatible in support of the large body of existing software. Some additional features have been retrofitted in a more awkward way than if the language had been developed with a clean slate. As a result, parts of the syntax have grown cryptic. More significantly, the flexibility given to a programmer for controlling low-level aspects comes with responsibility. Rather than one way to express something, there may be five alternatives. An experienced and knowledgeable developer can use this flexibility to pick the best alternative and improve the result. Yet both novice and experienced programmers can easily choose the wrong alternative, leading to less efficient or flawed software.

2 Programming Language Design

2.1 Interpreted versus Compiled

Chapter 1.3 of our book describes the distinction between *low-level* and *high-level* programming languages. At its core, a computing architecture supports an extremely limited set of data types and operations. For this reason, we describe a CPU's machine language as a *low-level* programming language. It is possible to develop software directly for that machine language, especially with applications for which execution speed is of utmost concern. However, it is extremely inconvenient to develop complex software systems in a low-level language. High-level programming languages were conceived to better support a programmer's expressiveness, thereby reducing the development time of software systems, providing greater opportunity for code reuse, and improving the overall reliability and maintainability of software. But, that high-level software must be translated back to a CPU's machine language in order to execute on that computer. This translation is typically automated in the form of an ***interpreter*** or ***compiler***.

Python is an example of an ***interpreted language***. We "run" a typical Python program by feeding its source code as input to another piece of software known as the Python interpreter. The Python interpreter is the software that is actually executing on the CPU, in effect adapting its outward behavior to match the semantics indicated by the given source code. The translation from high-level to low-level operations is performed on-the-fly[1].

In contrast, C++ is an example of a ***compiled language***. The translation of high-level source code to low-level machine code takes place in advance of the software being executed by the end user, relying on a distinct two-phase process. During the first phase ("compile-time"), the source code is fed as input to a special piece of software known as the ***compiler***. The compiler analyzes the source code based on the syntax of the language. If there are syntax errors, they are reported and the compilation fails. Otherwise, the compiler translates the high-level code into machine code for the computing system, generating a file known as an ***executable***. During the second phase ("run-time"), the executable is started on a computer (often by a user). We note that the compiled executable is catered to one particular machine language; different versions of the executable must be distributed for use on different computing platforms. Yet, the compiler and the original source code are not required to run the executable on a given computer (they would only be needed by the developer to regenerate a new executable if any change were to be made to the software).

The greatest advantage of the compilation model is execution speed. In essence, the more that can be handled at compile-time, the fewer CPU cycles are spent at run-time. By performing the full translation to machine code in advance, the execution of the software is streamlined so as to perform only those computations that are a direct part of the software application. A secondary advantage is that the executable can be distributed to customers as free-standing software (i.e., without need for an installed interpreter), and without exposing the original source code that was used to generate it (although some companies choose to "open source" their software).

The primary advantage of an interpreted language is greater platform-independence, as the primary source code can be widely distributed, with the platform-specific translations enacted by a locally-installed interpreter. A secondary advantage of interpreters is that they often support more dynamic language features, since analysis (or even modifcation) of code fragments can be performed at run-time, and they can serve a dual role as a development platform, providing the programmer with more robust feedback and interaction when problems arise.

2.2 Dynamic versus Static Typing

We noted that for compiled languages, there is an advantage in doing as much work as possible at compile-time so as to streamline the run-time process. It is this fact that motivates the single greatest distinction between Python and C++. Python is a *dynamically-typed* language. An identifier can be assigned to an underlying value, within a given scope, using an assignment statement such as

```
age = 42
```

We happen to know that `age` is being assigned to an integer value in this case, yet we did not make any syntactic declaration regarding the data type. That same identifier could later be reassigned to the string `'Stone'`. Types are not formally associated with the identifiers, but rather with the

[1]Technically, a small amount of compilation takes place in Python when parsing the source code. That portion of the translation results in a saved `.pyc` file that can be reused on a later execution to avoid re-parsing the code.

underlying objects (in effect, the value 42 "knows" that it is an integer). When identifiers are used in expressions, the legitimacy depends upon the type of the underlying object. The expression age + 1 will be valid when age is an integer yet invalid when age is a string. The method call age.lower( ) will be legal when age is a string yet illegal when age is an integer.

In Python, these expressions are evaluated at run-time. When encountering an expression such as age.lower( ), the interpreter determines whether the object currently associated with the name age supports the syntax lower( ). If so, the expression is evaluated successfully; if not, a run-time error occurs. The same principle of dynamic typing applies to the declaration of functions. The formal parameters in the signature serve as placeholders for the required number of actual parameters, yet there is no explicit statement of type; these identifiers are assigned to the actual objects sent by the caller at run-time. Class definitions also rely on dynamic typing for the data members, which are generally initialized in the constructor but never explicitly declared.

Python style of waiting until run-time to evaluate the legitimacy of commands is known as "duck typing" (if it walks like a duck and quacks like a duck, then for all intents and purposes it is a duck). This flexibility allows for various forms of polymorphism. For example, the sum function accepts a parameter that is assumed to be a sequence of numbers. It works whether that sequence is in the form of a **list**, a tuple, or a set, so long as the parameter is iterable. Yet Python also allows you to query the type of an object at run-time, allowing for another form of polymorphism. A function can vary its behavior based upon the type of an actual parameter. For example in Chapter 6.2 of our book we provided a Point.__mul__ implementation that specialized the multiplication semantics depending upon whether the actual parameter was a scalar value or another point.

C++ is a *statically-typed* language. An explicit type declaration is required for every identifier. The following demonstrates a type declaration followed by an assignment.

```
int age;
age = 42;
```

The first line is a declaration that establishes the identifier age as an integer value in the current scope, and the second line is an assignment statement that sets the value of the variable (it is also possible to initialize the value as part of a declaration statement; see Section 4.2 for details). Type declarations appear in many contexts, making explicit the type of parameters and return values for functions, and the type of data members stored by an instance of a class.

The reason for requiring programmers to make such declarations is that it allows for significantly more work to be done at compile-time rather than run-time. For example the legality of the subsequent assignment age = 42 is apparent at compile-time based upon knowledge of the data type. In similar spirit, if a programmer attempts to send a string to a function that expects a floating-point number, as in sqrt("Hello"), this error can be detected at compile-time. Type declarations also help the system in better managing the use of memory.

The choice between dynamically versus statically-typed languages is often (though not always) paired with the choice between interpreted and compiled languages. The primary advantage of static typing is the earlier detection of errors; this early detection is more significant with a compiled language, for which there is a distinction between compile-time errors and run-time errors. Even if static typing is used in a purely interpreted language, those errors will not arise until the program is executed. The primary advantages of dynamic typing are the reduced syntactical burden associated with explicit declarations together with more direct support for polymorphism.

3 A First Glance at C++

As our first example, Figure 1 presents a side-by-side view of corresponding Python and C++ programs. Both ask the user to enter two integers, computing and displaying their greatest common divisor. In this section, we highlight some difference between the two languages.

3.1 Superficial Differences

We first draw attention to the use of punctuation in C++ for delimiting the basic syntactic structure of the code. An individual command in Python (e.g., `u = v`) is followed by a newline character, designating the end of that command. In C++, each statement must be explicitly terminated with a semicolon. For example, we see the semicolon after the command `u = v` on line 10.

There is also a difference in designating a "block" of code. In Python, each block is preceded by a colon, with indentation subsequently used to designate the extent of a multi-line block. In C++, these blocks of code are explicitly enclosed in curly braces { }. The body of the while loop in the C++ version consists of everything from the opening brace at the end of line 8 until the matching right brace on line 12. That loop is itself nested within the function body that begins with the left brace on line 4 and concludes with the right brace on line 14.

Python

```
 1  def gcd(u, v):
 2    # we will use Euclid's algorithm
 3    # for computing the GCD
 4    while v != 0:
 5      r = u % v      # compute remainder
 6      u = v
 7      v = r
 8    return u
 9
10  if __name__ == '__main__':
11    a = int(raw_input('First value: '))
12    b = int(raw_input('Second value: '))
13    print 'gcd:', gcd(a,b)
```

C++

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int gcd(int u, int v) {
 5    /* We will use Euclid's algorithm
 6       for computing the GCD */
 7    int r;
 8    while (v != 0) {
 9      r = u % v;    // compute remainder
10      u = v;
11      v = r;
12    }
13    return u;
14  }
15
16  int main( ) {
17    int a, b;
18    cout << "First value: ";
19    cin >> a;
20    cout << "Second value: ";
21    cin >> b;
22    cout << "gcd: " << gcd(a,b) << endl;
23    return 0;
24  }
```

Figure 1: Programs for computing a greatest common divisor, as written in Python and C++.

For the most part, the use of whitespace is irrelevant in C++. Although our sample code is spaced with one command per line and with indentation to highlight the block structure, these are not formal requirements of the language syntax. For example, the definition of the gcd function could technically be expressed in a single line as follows:

```
int gcd(int u, int v) { int r; while (v != 0) { r = u % v; u = v; v = r; } return u; }
```

To the compiler, this is the same definition as our original. Of course, to a human reader this version is nearly incomprehensible. So as you transition from Python, we ask that you continue using whitespace to make your source code legible.

This first example demonstrates a few other superficial differences. C++ requires the boolean condition for a while loop to be expressed within parentheses (see line 8). We do not need to do so in Python (see line 4), although parentheses can be used optionally. We also see a difference in the symbols used when providing inlined comments. In Python, the # character is used to designate the remainder of the line as a comment. Two different commenting styles are allowed in C++. An inlined comment is supported using the // pattern, as seen at line 9. Another style is demonstrated on lines 5 and 6, starting with the /* pattern, and ending with the */ pattern. This style is particularly convenient as it can span multiple lines of source code.

3.2 Static Typing

The more significant differences between the Python and C++ versions of our example involve the distinction between ***dynamic*** and ***static*** typing, as we originally discussed in Section 2.2. Even in this simple example, there are three distinct manifestations of static typing. The formal parameters (i.e., identifiers u and v) are declared without any explicit type designation in the Python signature at line 1. In the corresponding declaration of parameters in the C++ signature, we find explicit type declaration for each parameter within the syntax gcd(**int** u, **int** v). This information serves two purposes for the compiler. First, it allows the compiler to check the legality of the use of u and v within the function body. Second, it allows the compiler to enforce that integers be sent when the function is called (as at line 22).

The second manifestation of static typing is the explicit designation of the *return type* as part of a formal signature in C++. In line 4 of our C++ example, the declaration **int** at the beginning of the line labels gcd as a function that returns an integer. Again, the compiler uses this designation to check the validity of our own code (namely that we are indeed returning the correct type of information at line 13), as well as to check the caller's use of our return value. For example, if the caller invokes the function as part of an assignment g = gcd(54,42), this would be legal if variable g has been declared as an integer or compatible numeric type, yet illegal if g were a string.

Finally, we note the declaration of variable r at line 7 of our C++ code. This designates r as a local variable representing an integer, allowing its use at lines 9 and 11. Had we omitted the original declaration, the compiler would report an error "cannot find symbol" regarding the later use of r (the analog of a NameError in Python). Formally, a declared variable in C++ has scope based upon the most specific set of enclosing braces at the point of its declaration. In our original example, the variable r has scope as a local variable for the duration of the gcd function body (as is also the case with our Python version). Technically, since this variable's only purpose is for temporary storage during a single invocation of the while loop body, we could have declared it within the more narrow scope of the loop body (Python does not support such a restricted scope).

3.3 Input and Output

Python and C++ differ greatly in the techniques used to support the gathering of input and production of output. We will discuss this topic more thoroughly in Section 6, but for now we examine the use of input and output in our first example from Figure 1. In both programs, the goal is to read two integers and then to display the result.

We begin by examining the corresponding commands that display the result. In Python, the built-in **print** command is used at line 13 to display the string `'gcd:'` followed by the return value of the function call `gcd(a,b)`. We rely on the fact that the `print` command automatically inserts an extra space between the two arguments and a newline character after the final argument.

In C++, input and output is generally managed through an abstraction known as a "stream." The definition of a stream must be explicitly imported from a standard library. In our first C++ example, lines 1 and 2 are used to load the necessary definitions into our context. The actual display of the result is accomplished by line 22 of that code. The **cout** identifier represents a special output stream used to display information to the user console. The << symbol is an operator for inserting data into that output stream. In this case we insert the string `"gcd: "`, followed by the return value of expression `gcd(a,b)`, followed by the identifier **endl** which represents a newline character. In contrast to our Python code, we are responsible for explicitly including the separating space as part of our initial string literal, and for inserting the endline character.

For gathering input, our Python version uses the **raw_input** function at lines 11 and 12. Each call to that function prompts the user and then reads a single line of input from the keyboard. The result is returned in the form of a character string. Because we want to interpret that response as an integer, we explicitly use the **int**(...) syntax to construct an integer based upon the parsed string. In C++, we use the **cin** object to read input from the user console. We must display a prompt separately, as seen at line 18 (note the lack of an endline character). We then read an integer from the user at line 19 with the command **cin** >> a. The >> operator is used to extract information from the stream into a variable. Here, we see an advantage of C++'s static typing. We do not need to explicitly convert the read characters into an integer. The conversion is implicit because variable a was already declared to have type **int**.

3.4 Executing a Program

Finally, we examine the treatment of the overall program and the different language models for executing code. Starting with Python, let us assume that the source code is saved in a file `gcd.py`. The Python program is executed by starting the Python interpreter while designating the source code file. For example, if working directly with the operating system, we might issue the command

```
python gcd.py
```

The interpreter then begins reading commands from the source code. In the case of our example, the interpretation of lines 1–8 results in the definition of (but not the calling of) the `gcd` function. The interpreter continues by executing the body of the **if** `__name__` == `'__main__'` conditional at lines 10–13. Technically, we could have written this script omitting the conditional at line 10, with the subsequent commands at the top-level context. The advantage of the given style is that it allows us to differentiate between times when this file is started as the primary source code and when it is imported as a module from some other context. If the command **import** `gcd` were executed from another context, the definition of the `gcd` function would be loaded, but lines 11–13 would be bypassed. This special conditional can be used as a *unit test* when developing a module that is part of a larger system.

In C++, source code must first be compiled. A popular compiler is distributed by `gnu.org` and typically installed on a system as a program named `g++`. If our source code were saved in a file named `gcd.cpp`, the compiler could be invoked from the operating system with the command,

```
g++ -o gcd gcd.cpp
```

The compiler will report any syntax errors that it finds. If all goes well it produces a new file named `gcd` (or `gcd.exe` with the Windows operating system) that is an *executable*. It can be started on the computer just as you would start any other executable. It is also possible to compile C++ code using an integrated development environment (akin to Python's IDLE). An IDE typically relies upon the same underlying compiler, but provides more interactive control of the process.

The flow of control for the execution of a C++ program begins with an implicit call to the function named main. We see this function definition starting at line 16 of our sample C++ source code. The **int** return value for main is a technical requirement. The value is returned to the operating system at the conclusion of the program. It is up to the operating system to interpret that value, although zero historically indicates a successful execution while other values are used as error codes (such a return value can be specified in Python as a parameter to the sys.exit function).

As a final comment, we note that the use of a main function in C++ is not quite the same as the **if** __name__ == '__main__' construct in Python. We discussed how the Python technique could be used to provide a unit test that would be executed when the file is the primary source code, but ignored when that module was imported from elsewhere. When a project is implemented with multiple source files in C++, the compiler requires that precisely one of them has a main routine. As a consequence, the gcd function as provided in our sample `gcd.cpp` file could not be used as part of a larger project (because there would be conflicting definitions for main). With a more typical C++ style, such a utility function would be provided in a file without a main function, and imported as needed by other applications. We will discuss the development of multifile projects in Section 12.

4 Data Types and Operators

4.1 Primitive Data Types

Figure 2 provides a summary of common primitive data types in C++, noting the correspondence to Python's types. We emphasize the following aspects of that comparison.

Boolean values

The logical **bool** type is supported by both languages, although the literals **true** and **false** are uncapitalized in C++ while capitalized in Python. In both languages, boolean values are stored internally as integers, with false represented using value 0, and true represented as 1.

Varying precision of numeric types

C++ offers the programmer more fine-grained control in suggesting the underlying precision when storing numbers. There exist three fixed-precision integer types: **short**, **int**, and **long**. However, the precise number of bits devoted to these types is system-dependent, with typical values shown in Figure 2. Python's **int** type is usually implemented with the precision of a C++ **long**.

Each of the C++ integer types has an **unsigned** variant that is constrained to represent non-negative numbers. For example, a variable can be declared with type **unsigned short**. Whereas a (signed) integer type with b bits has a typical range from $-(2^{b-1})$ to $+(2^{b-1}-1)$, a corresponding

C++ Type	Description	Literals	Python analog
bool	logical value	**true** **false**	**bool**
short	integer (often 16 bits)		
int	integer (often 32 bits)	39	
long	integer (often 32 or 64 bits)	39L	**int**
——	integer (arbitrary-precision)		**long**
float	floating-point (often 32 bits)	3.14f	
double	floating-point (often 64 bits)	3.14	**float**
char	single character	`'a'`	
string[a]	character sequence	`"Hello"`	**str**

Figure 2: The most common primitive data types in C++.

[a]Not technically a built-in type; included from within standard libraries.

unsigned type would have range from 0 to $2^b - 1$. The greater range of positive numbers can be used for contexts when a value cannot be negative (such as when describing the size of a container).

C++ also supports two different floating-point types, **float** and **double**, with a **double** historically represented using twice as many bits as a **float**. In C++, the **double** is most commonly used and akin to what is named **float** in Python.

Finally, we note that Python's **long** type serves a completely different purpose, representing integers with *unlimited* magnitude. There is no such standard type in C++ (although some C++ packages for arbitrary-precision integers are distributed independently).

Character strings

C++ supports two different types for representing text. The **char** type provides an efficient representation of a single character of text, while the **string** class serves a purpose similar to Python's **str** class, representing a sequence of characters (which may happen to be an empty string or a single-character string). To distinguish between a **char** and a one-character string, a **string** literal must be designated using double quote marks (as in `"a"`). The use of single quotes is reserved for a **char** literal (as in `'a'`). An attempt to misuse the single-quote syntax, as in `'impossible'`, results in a compile-time error.

From a technical point of view, the **string** class is not a built-in type; it must be included from among the standard C++ libraries. Although the formal methods of the C++ **string** class are not the same as the Python **str** class, many behaviors are common. However, in contrast with Python's immutable **str** class, a C++ **string** is *mutable*. So the expression s[index] can be used to access a particular character, or to alter that character with an assignment s[index] = newChar. There is a similar discrepancy between the C++ syntax s.append(t), which mutates instance s by appending the contents of string t, and the syntax s+t, which produces a concatenation as a third string, leaving the two original strings unchanged. A summary of the most commonly used string operations is given in Figures 3 and 4, with Figure 3 describing the nonmutating behaviors and Figure 4 describing the mutating behaviors.

Syntax	Semantics

s.size( )
s.length( )
	Either form returns the number of characters in string s.

s.empty( )	Returns **true** if s is an empty string, **false** otherwise.
s[index]	Returns the character of string s at the given index (unpredictable when index is out of range).
s.at(index)	Returns the character of string s at the given index (throws exception when index is out of range).
s == t	Returns **true** if strings s and t have same contents, **false** otherwise.
s < t	Returns **true** if s is lexicographical less than t, **false** otherwise.
s.compare(t)	Returns a negative value if string s is lexicographical less than string t, zero if equal, and a positive value if s is greater than t.

s.find(pattern)
s.find(pattern, pos)
	Returns the least index (greater than or equal to index pos, if given), at which pattern begins; returns **string**::npos if not found.

s.rfind(pattern)
s.rfind(pattern, pos)
	Returns the greatest index (less than or equal to index pos, if given) at which pattern begins; returns **string**::npos if not found.

s.find_first_of(charset)
s.find_first_of(charset, pos)
	Returns the least index (greater than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.

s.find_last_of(charset)
s.find_last_of(charset, pos)
	Returns the greatest index (less than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.

s + t	Returns a concatenation of strings s and t.
s.substr(start)	Returns the substring from index start through the end.
s.substr(start, num)	Returns the substring from index start, continuing num characters.
s.c_str( )	Returns a C-style character array representing the same sequence of characters as s.

Figure 3: Nonmutating behaviors supported by the **string** class in C++.

Syntax	Semantics
s[index] = newChar	Mutates string s by changing the character at the given index to the new character (unpredictable when index is out of range).
s.append(t)	Mutates string s by appending the characters of string t.
s += t	Same as s.append(t).
s.insert(index, t)	Inserts copy of string t into string s starting at the given index.
s.insert(index, num, c)	Inserts num copies of character c into string s starting at the given index.
s.erase(start)	Removes all characters from index start to the end.
s.erase(start, num)	Removes num characters, starting at given index.
s.replace(index, num, t)	Replace num characters of current string, starting at given index, with the first num characters of t.

Figure 4: Mutating behaviors supported by the **string** class in C++.

Arrays

The standard structure for storing a mutable sequence of values in Python is the **list** class. This class provides many convenient behaviors for processing the sequence, and it provides support for seemlessly expanding the size of the sequence as needed. The standard C++ libraries, which we will discuss further in Section 10, includes a **vector** class with similar properties.

However, C++ also supports a more low-level sequence, known as an ***array***, which has its origin in the C programming language (in fact all of the built-in types for C++ were defined for C). An array is a contiguous chunk of memory used to store a data sequence. What makes an array different from a structure such as a Python **list** is that the size of the array must be fixed when the array is constructed and that the contents of the array must have the same data type (because the values are stored directly in the array, rather than referentially). As with Python, C++ arrays are zero-indexed and rely on the syntax of square brackets for indexing. For example, if measurements is a variable representing an array of **double** values, then the expression measurements[7] is used to access the **double** at index 7 of the sequence (that is, the eighth entry).

Finally, we note that in the earlier C language, which does not have a **string** class, a character sequence is represented directly as an array of **char** values. In C++, the **string** class relies upon an underlying character array for storage, but provides more robust support for convenient operations. That said, we note that the c_str( ) method of the string class produces a C-style character array for a string instance, as is sometimes needed when working with legacy code.

4.2 Declarations and Initialization

We have emphasized that C++ is a statically-typed language and that the type of each variable must be explicitly declared before use. We saw several examples of type declarations in our first glance of C++, such as the declaration of local variable r within the gcd function at line 7 of Figure 1,

```
int r;
```

It is possible to declare several variables of the same type in a single declaration. For example, line 17 of that same program declared the variables a and b within the main function as follows,

```
int a,b;
```

A declaration alerts the C++ compiler as to the type of data that will be stored by the named variable. This allows it to verify subsequent syntax at compile-time. Knowledge of the data type also allows the system to reserve the appropriate amount of memory for representing an instance of the type. It is important to note that the system does not necessarily *initialize* the variable. For primitive types, the initial value is indeterminate, as it is based upon the previous setting of the newly reserved bits of memory. While it could be initialized using a separate assignment statement, we prefer the use of what we term the ***constructor*** syntax when the initial value is known at the time of the declaration. An example of the syntax is

```
int age(42);
```

This declares a new integer variable age, yet initializes it to have the indicated value 42. When using this syntax, we can use any valid expression to designate the initial value, such as,

```
int age(curYear − birthYear);
```

We can initialize multiple variables of a common type in a single declarations, as in

```
int age(42), zipcode(63103);        // two new variables
```

Although we noted that declared variables of primitive types are not automatically initialized, a declared variable of a *class* type will be initialized by automatically invoking a form of that class's constructor. Using the **string** class as an example, consider the following three declarations.

```
string response;              // guaranteed to be the empty string ""
string greeting("Hello");     // initialized to "Hello"
string rating(3, 'A');        // initialized to "AAA"
```

The first version invokes what is known as the *default constructor* for the class. This is a zero-parameter version of the constructor, which in the case of strings produces an empty string. The second of these lines formally invokes the constructor with a single parameter, the character sequence `"Hello"`. The third example invokes a two-parameter form of the **string** constructor, resulting in a sequence of n consecutive copies of a given character.

Arrays

If the desired size of an array is known at compile time, it can be declared using a syntax such as

```
double measurements[300];
```

This declares the name measurements to be an array of doubles and causes the system to allocate memory for storing precisely 300 entries. However, the values of the individual entries are indeterminate (just as when declaring a single **double**). Typically, the declaration of such an array might be followed by a loop to initialize the entries to meaningful values. Yet it is possible to initialize values of an array as part of the declaration, using a syntax such as the following.

```
int daysInMonth[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Notice that we did not explicitly give the size of the array between the square brackets; it will be implicitly set based on the number of indicated entries. If we had given an explicit size for the array that was larger than the indicated initialization list, the beginning of the array will be filled in using the specified values and the rest is set to zero. In the special case of an array of characters, it is possible to initialize the array using a literal, as follows.

```
char greeting[ ] = "Hello";
```

As a technicality, this becomes an array with size 6 because all C-style character sequences are explicitly terminated with an extra zero-value to designate the end of the sequence.

In the case of class types, the declaration of an array causes not only the allocation of memory but the default initialization of each individual entry. For example, in the following declaration all entries of the array are guaranteed to be initialized to empty strings.

```
string messages[20];
```

Thus far, our declarations have assumed that the size of an array is known at compile time. An approach for dynamically allocating arrays at run-time will be presented in Section 8.5.

Mutability

Python and C++ have very different approaches to the concept of mutability. For built-in types, Python makes a clear distinction between which are mutable and which are immutable. For example, it offers a **list** class for representing *mutable* sequences, and a **tuple** class for representing *immutable* sequences (a similar discrepancy exists between the **set** and **frozenset** classes).

C++ takes a different approach. All types are assumed to be mutable, but particular instances can be designated as immutable by the programmer. Furthermore, the immutability is strictly enforced by the compiler. Syntactically, the immutability is declared using the keyword **const** in specific contexts. For example, a new variable can be declared as a constant, as in

```
const int age(42);      // immortality
```

With this declaration, any subsequent attempt to change that value results in a compile-time error (e.g., age++). While a programmer may declare a non-const variable yet leave its value unchanged, the explicit declaration of **const** in this context allows the compiler to better optimize the program. It also serves as a meaningful label for another programmer who is reading the code. More significant uses of the **const** keyword arise when describing the treatment of parameters in function signatures, or the effect of method calls upon the state of an object. We will discuss those uses in later sections.

4.3 Operators

Figure 5 provides a comparison between the operators supported by Python and C++. The operators are largely the same between the two languages, but there are some notable discrepancies.

Numeric types

For numeric values, Python differentiates between ***true division*** (i.e., /), ***integer division*** (i.e., //), and ***modular arithmetic*** (i.e., %), as originally discussed in Chapter 2.4 of our book. C++ supports operators / and %, but not //; in fact, we already saw that symbol used to designate inline comments in C++. The semantics of the C++ / operator depends upon the type of operands. When both operands are integral types, the result is the integer quotient; if one or both of the operands are floating-point types, true division is performed. To get true division with integral types, one of the operands must be explicitly cast to a float (see Section 4.4).

Both Python and C++ support an operator-with-assignment shorthand for most binary operators, as with x += 5 as a shorthand for x = x + 5. Yet, C++ supports an additional ++ operator for the common task of incrementing a number by one. In fact, there are two distinct usages known as ***pre-increment*** (e.g., ++x) and ***post-increment*** (e.g., x++). Both of these add one to the value of x, but they can be used differently in the context of a larger expression. For example, if indexing a sequence, the expression groceries[i++] retrieves the entry based upon the original index i, yet subsequently increments that index. In contrast, the syntax groceries[++i] causes the value of the index to be incremented *before* accessing the associated entry of the sequence. Similar support exists for decrementing a value by one with the −− operator. The pre- and post- versions of these operators are valuable tools for an experienced programmer, but their use leads to subtle code and potential mistakes. We recommend that they be used in isolated contexts until mastered.

	Python	C++	Description
	Arithmetic Operators		
	−a	−a	(unary) negation
	a + b	a + b	addition
	a − b	a − b	subtraction
	a * b	a * b	multiplication
▷	a ** b		exponentiation
	a / b	a / b	standard division (depends on type)
▷	a // b		integer division
	a % b	a % b	modulus (remainder)
▷		++a	pre-increment operator
▷		a++	post-increment operator
▷		−−a	pre-decrement operator
▷		a−−	post-decrement operator
	Boolean Operators		
▷	**and**	&&	logical and
▷	**or**	||	logical or
▷	**not**	!	logical negation
▷	a **if** cond **else** b	cond ? a : b	conditional expression
	Comparison Operators		
	a < b	a < b	less than
	a <= b	a <= b	less than or equal to
	a > b	a > b	greater than
	a >= b	a >= b	greater than or equal to
	a == b	a == b	equal
▷	a < b < c	a < b && b < c	chained comparison
	Bitwise Operators		
	~a	~a	bitwise complement
	a & b	a & b	bitwise and
	a | b	a | b	bitwise or
	a ^ b	a ^ b	bitwise XOR
	a << b	a << b	bitwise left shift
	a >> b	a >> b	bitwise right shift

Figure 5: Python and C++ operators, with differences noted by ▷ symbol.

Boolean operators

While Python uses the words **and**, **or**, and **not** for the basic logical operators, C++ relies on the respective symbols &&, ||, and ! (although some C++ compilers now support such named operators). The history of those symbols dates back to the C language. It is important not to confuse the logical operators && and || with the bitwise operators & and |.

Another pitfall for Python programmers converting to C++ is use of a syntax such as a < b < c for numeric types. In Python, operators can be chained in such an expression that is true if both a < b and b < c. In C++, this logic must be expressed as a < b && b < c. The more significant problem is that a < b < c is *legal* C++ syntax, yet with unexpected semantics. This is parsed as (a < b) < c. The boolean expression (a < b) evaluates to either **false** or **true**. However, that result will be coerced into the integer 0 or 1 for the comparison with c. The result of the full expression then depends upon whether c is greater than that 0 or 1 value.

Class types

By default, most operators cannot be used with instances of a class type. However C++ allows the author of a class to provide specialized semantics for operators, when desired. As an example, the **string** class overloads the + operator so that the expression s + t results in a new string that is the concatenation of existing strings s and t. Yet the expression s * 3 is illegal when s is a string, because no such behavior is defined for the C++ **string** class (in contrast to Python, which supports such an operator for the **str** class). The syntax for defining overloaded operators in C++ will be introduced in Section 7.2, when we discuss the robust version of our user-defined Point class.

4.4 Converting Between Types

In Python, we saw several scenarios in which implicit type conversion is performed. For example, when performing the addition 1.5 + 8 the second operand is coerced into a floating-point representation before the addition is performed.

There are similar settings in which C++ implicitly casts a value to another type. Because of the static typing, additional implicit casting may take place when assigning a value of one type to a variable of another. Consider the following example:

```
int a(5);
double b;
b = a;                    // sets b to 5.0
```

The final command causes b to store a floating-point representation of the value 5.0 rather than the integer representation. This is because variable b was explicitly designated as having type **double**. We can also assign a **double** value to an **int** variable, but such an implicit cast may cause the loss of information, as any fractional portion of that value will be truncated.

```
int a;
double b(2.67);
a = b;                    // sets a to 2
```

There are other scenarios in which C++ implicitly converts between types that would not normally be considered compatible. Some compilers will issue a warning to draw attention to such cases, but there is no guarantee.

On a related note, there are times when we want to force a type conversion that would not otherwise be performed. Such an ***explicit cast*** is done using a syntax similar to Python, where the name of the target type is used as if a function.

```
int a(4), b(3);
double c;
c = a/b;              // sets c to 1.0
c = double(a)/b;      // sets c to 1.33
```

The first assignment to c results in 1.0 because the coercion to a **double** is not performed until after the integer division a/b is performed. In the second example, the explicit conversion of a's value to a **double** causes a true division to be performed (with b implicitly coerced). However, we cannot use this casting syntax to perform all type conversions. For example, we *cannot* safely mimic Python's approach for converting a number to a string, as with **str**(17), or to convert a string to the corresponding number, as with **int**("17"). Unfortunately, conversions back and forth between strings require more advanced techniques. We will discuss one such approach in Section 6.6 using an object known as a stringstream.

5 Control Structures

5.1 While Loops

We demonstrated an example of a **while** loop as part of our first example in Section 3. The logic is similar to Python, but with superficial differences in syntax. Most notably, parentheses are required around the boolean condition in C++. In that first example, curly braces were used to delimit the commands that comprise the body of the loop. Technically those braces are only needed when the body uses two or more distinct statements. In the absence of braces, the next single command is assumed to be the body.

C++ also supports a **do-while** syntax that can be a convenient remedy to the "loop-and-a-half" problem, as seen with while loops on page 165 of our book. Here is a similar C++ code-fragment for requesting a number between 1 and 10, repeating until receiving a valid choice:

```
  int number;
  do {
    cout << "Enter a number from 1 to 10: ";
    cin >> number;
  } while (number < 1 || number > 10);
```

Please note that we have not properly handled the exceptional case when a noninteger is entered.

5.2 Conditionals

A basic **if** statement is quite similar in style, again requiring parentheses around the boolean condition and curly braces around a compound body. As a simple example, here is a construct to set a number to its absolute value.

```
if (x < 0)
  x = −x;
```

Notice that we did not need braces for a body with one command.

C++ does not use the keyword elif for nesting conditionals, but it is possible to nest a new **if** statement within the body of an **else** clause. Furthermore, a conditional construct is treated syntactically as a single command, so nesting does not require excessive braces. Our first example of a nested conditional in Python was given on page 144 of Chapter 4.4.2 of our book. That code could be written in C++ to mimic Python's indentation as follows (assuming groceries is an adequate container):

```
if (groceries.length( ) > 15)
  cout << "Go to the grocery store" << endl;
else if (groceries.contains("milk"))
  cout << "Go to the convenience store" << endl;
```

5.3 Nonboolean Expressions as Conditions

On page 142 of our book, we demonstrate ways in which nonboolean data types can be used in place of a boolean expression for a conditional statement. C++ also provides support for coercing certain nonboolean data types in certain contexts. With built-in numeric types, C++ treatment is similar to Python's, in that a zero value is coerced as **false** and any nonzero value as **true**. So we might write the following code, assuming that mistakeCount is declared as an integer.

```
if (mistakeCount)                    // i.e., if (mistakeCount != 0)
  cout << "There were " << mistakeCount << " errors" << endl;
```

For class types, C++ allows the author of the class to determine whether values of that type can be coerced into a **bool** (or any other primitive type, for that matter). So it would be possible to define a container type that mimics Python's treatment, with empty containers treated as **false** and nonempty containers treated as **true**. We should note, however, that neither the standard container classes or strings support such coercion in C++.

The implicit conversion from numbers to booleans in C++, together with the treatment of an assignment statement, is to blame for a common pitfall exemplified by the falling errant code.

```
double gpa;
cout << "Enter your gpa: ";
cin >> gpa;
if (gpa = 4.0)
  cout << "Wow!" << endl;
```

Do you see the problem? The mistake is the use of the assignment operator (gpa = 4.0) rather than the equivalence operator (gpa == 4.0). The above code is syntactically valid in C++, but it does not behave as you might expect. Rather than comparing the inputted value of gpa, that statement reassigns variable gpa the value of 4.0, essentially overwriting the response given by the user. Furthermore, C++ considers the value of the assignment expression in the larger context to be the newly assigned value, which itself is implicitly coerced to a boolean value. Thus, no matter what the user enters, the gpa is reset to 4.0, coerced to **true**, and the conditional body is entered.

It is natural to ask why the designers of C and C++ allow such a behavior. One reason is to support the chaining of assignments using the following syntax.

```
a = b = 4.0;          // assuming both a and b were previously declared
```

C++ treats this as two separate operations, evaluated from right-to-left, as if written as

```
a = (b = 4.0);
```

So b is assigned the value 4.0, and then the new value of b serves as the result of the parenthesized subexpression. This allows a to be subsequently assigned.

To avoid such a common pitfall, Python disallows use of an assignment statement in the context of a conditional. Code such as **if** (gpa = 4.0) results in a syntax error. Python support the chaining syntax a = b = 4.0 using a different mechanism, just as it does with inequalities like a < b < c.

5.4 For Loops

C++ supports a **for** loop, but with very different semantics than Python's. The style dates back to its existence in C, to provide a more legible form of the typical ***index-based*** loop pattern described in Chapter 4.1.1 of our book. An example of a loop used to count downward from 10 to 1 is as follows:

```
for (int count = 10; count > 0; count−−)
  cout << count << endl;
cout << "Blastoff!" << endl;
```

Within the parentheses of the for loop are three distinct components, each separated by a semicolon. The first is an *initialization* step that is performed once, before the loop begins. The second portion is a *loop condition* that is treated just as a loop condition for a while loop; the condition is tested before each iteration, with the loop continuing while true. Finally we give an *update* statement that is performed automatically at the end of each completed iteration. The for loop syntax is just a convenient alternative to a while loop that better highlights the logic in some cases. The previous example is essentially identical in behavior to the following version:

```
int count = 10;                    // initialization step
while (count > 0) {                // loop condition
  cout << count << endl;
  count−−;                         // update statement
}
cout << "Blastoff!" << endl;
```

The for loop is far more general. It is possible to express *multiple* initialization or update steps in a for loop. This is done by using *commas* to separate the individual statements (as opposed to the semicolon that delimits the three different components of the syntax). For example, the sum of the values from 1 to 10 could be computed by maintaining two different variables as follows:

```
int count, total;
for (count = 1, total = 0; count <= 10; count++)
  total += count;
```

It is also possible to omit the initialization or update steps, so long as the semicolons remain as separators. The loop condition is the only strictly required component.

5.5 Defining a Function

Our initial C++ example from Section 3 includes the definition of a gcd function. We emphasized the need to explicitly designate the type of each individual parameter as well as the return type. Together with the name of the function, this information is collectively known as the ***signature*** of the function. For example, the signature in our original example was **int** gcd(**int** u, **int** v). This signature serves as a guide for a potential caller of the function, and it provides sufficient information for the compiler to enforce proper usage.

Some functions do not require parameters and some do not provide a return value. Without any parameters, a function definition still requires opening and closing parentheses after the function name. Functions that do not provide a return value must still designate so by using a special keyword **void** in place of a return type. As an example, the following function prints a countdown from 10 to 1. It does not accept any parameters nor return any value.

```
void countdown( ) {
  for (int count = 10; count > 0; count−−)
    cout << count << endl;
}
```

We used an alternative version of this function in Chapter 5.2.2 of our book, to demonstrate the use of optional parameters in Python. The same technique can be used in C++, with the syntax

```
void countdown(int start=10, int end=1) {
  for (int count = start; count >= end; count−−)
    cout << count << endl;
}
```

This signature will support a calling syntax such as countdown(5,2) to go from 5 to 2, countdown(8) to go from 8 to 1, or countdown( ) to go from 10 to 1. There is a technical distinction between Python and C++ regarding the instantiation of a default parameter. In Python, the default parameter value is instantiated once upon declaration of the function, and then used when the function is called. This is occasionally significant, especially when using mutable objects as default values. In C++, default parameter values are (re)instantiated as needed with each call to the function.

6 Input and Output

Input and output can be associated with a variety of sources within a computer program. For example, input can come from the user's keyboard, can be read from a file, or transmitted through a network. In similar regard, output can be displayed on the user's screen, written to a file, or transmitted through a network. To unify the treatment of input and output, C++ relies on a framework of classes to support an abstraction known as a "stream." We insert data into an output stream to send it elsewhere, or extract data from an input stream to read it. A stream that provides us with input is represented using the istream class, and a stream that we use to send output elsewhere is represented using the ostream class. Some streams can serve as both input and output (iostream). In this section, we provide an overview of the most commonly used stream classes. A summary of those is given in Figure 6.

Class	Purpose	Library
istream	Parent class for all input streams	<iostream>
ostream	Parent class for all output streams	<iostream>
iostream	Parent class for streams that can process input and output	<iostream>
ifstream	Input file stream	<fstream>
ofstream	Output file stream	<fstream>
fstream	Input/output file stream	<fstream>
istringstream	String stream for input	<sstream>
ostringstream	String stream for output	<sstream>
stringstream	String stream for input and output	<sstream>

Figure 6: Various input and output stream classes.

6.1 Necessary Libraries

Technically, streams are not automatically available in C++. Rather, they are included from one of the standard libraries. A C++ library serves a similar purpose to a Python module. We will discuss them more fully in Section 12. In our initial example, we loaded the definitions as follows.

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

The first of these statements imports the iostream library (short for "input/output stream"). The second brings the definitions into our default namespace. In addition to the basic class definitions, this library defines two special instances for handling input to and from the standard console. **cout** (short for "console output") is an ostream instance used to print messages to the user, and **cin** (short for "console input") is an istream instance that reads input from the keyboard.

6.2 Console Output

Output streams support the << operator to insert data into the stream, as in **cout** << `"Hello"`. The << symbol subliminally suggests the flow of data, as we send the characters of `"Hello"` into the stream. As is the case with print in Python, C++ will attempt to create a text representation for any nonstring data inserted into the output stream. Multiple items can be inserted into the stream by repeated use of the operator, as in **cout** << `"Hello "`<< person << `". How are you?"`. Notice that we explicitly insert spaces when desired, in contrast to use of Python's print command. We must also explicitly output a newline character when desired. Although we could directly embed the escape character `\n` within a string, C++ offers the more portable definition of a special object **endl** that produces a newline character when inserted into the stream. To demonstrate typical usage patterns, Figure 7 provides several side-by-side examples in Python and C++.

6.3 Formatted Output

In Python, string formatting can be used to generate output in a convenient form, as in

```
print '%s: ranked %d of %d teams' % (team, rank, total)
```

Python

```
print "Hello"
print                       # blank line
print "Hello,", first
print first, last           # automatic space
print total
print str(total) + "."      # no space
print "Wait...",            # space; no newline
print "Done"
```

C++

```
cout << "Hello" << endl;
cout << endl;                    // blank line
cout << "Hello, " << first << endl;
cout << first << " " << last << endl;
cout << total << endl;
cout << total << "." << endl;
cout << "Wait... ";          // no newline
cout << "Done" << endl;
```

Figure 7: Demonstration of console output in Python and C++. We assume that variables first and last have previously been defined as strings, and that total is an integer.

The use of the % sign in this context is designed to mimic a long-standing routine named printf, which is part of the C programming language. Since C++ is a direct descendant of C, that function is available through a library, but it is not the recommended approach for C++ because it does not inherently support non-primitive data types (e.g., the C++ **string** class). Instead, formatted output can be generated through the output stream. Since data types are automatically converted to strings, the above example can be written in C++ as

```
cout << team << ": ranked " << rank << " of " << total << " teams" << endl;
```

More effort is needed to control other aspects of the formatting, such as the precision for floating-point values. As an example, assume that the variable pi holds the value 3.14159265. In Python, the expression `'pi is %.3f'`% pi produces the result `'pi is 3.142'`. The closest equivalent to the display of variable pi in C++ might be accomplished as

```
cout << "pi is " << fixed << setprecision(3) << pi << endl;
```

This command would result in the output `pi is 3.142`. The objects fixed and setprecision(3) are known as *manipulators* and must be included from the <iomanip> library. When inserted into the output stream with << they affect the format for subsequent values. In this particular case, the fixed manipulator says to print floating-point numbers with trailing decimal digits even when zero, and not to use scientific notation even for very large or very small values. The setprecision(3) manipulator specifies the number of digits beyond the decimal point to be used in the fixed format.

The minimum width of displayed values and the justification within that width can be controlled by additional manipulators. As an example, we might print an entry on a receipt as

```
cout << setw(10) << item << " " << setw(5) << quantity << endl;
```

This is equivalent to the Python command print `'%10s %5d'`% (item, quantity). If we execute this command once with values pencil and 50, and then with values pen and 100, the output is aligned as:

```
    pencil    50
       pen   100
```

As is the case with Python, data in a fixed-width field will be right-justified by default. We can switch to left-justification in C++ by inserting the manipulator left into the stream. For example, if we repeat the previous exercise using the command

```
cout << left << setw(10) << item << " "<< right << setw(5) << quantity << endl;
```

we get a result of

```
pencil       50
pen         100
```

It is worth noting that the C++ manipulators are different than Python's formatting tools, in that most of the manipulators we have demonstrated change the state of the output stream object. For example, once the fixed manipulator has been inserted into a stream, all subsequent numbers for that stream will be displayed in that format (unless a conflicting manipulator such as scientific is subsequently inserted). In our most recent example, notice that we explicitly inserted the manipulator right before outputting quantity to re-establish right-justification; otherwise, the formatting would still be affected by the earlier insertion of left. However, there are some exceptions to this rule. The setw manipulator, demonstrated above, only affects the next piece of displayed output. In our example, the insertion of setw(10) affected the display of the string item, yet it did not dictate a minimum width for the subsequent string `" "`.

6.4 Console Input

In Python, input from the console is typically read using the **raw_input** function, as in

```
person = raw_input('What is your name?)
```

In C++, keyboard input is managed through an input stream named **cin** (just as console output is managed by **cout**). The behavior of the Python example can be replicated with a C++ function named getline. Its calling syntax appears as

```
string person;
cout << "What is your name? ";       // prompts the user (without a newline)
getline(cin, person);                 // stores result directly in variable 'person'
```

Both Python's **raw_input** and C++'s getline read and remove all characters from the input stream up to and including the next newline, yet with the newline itself omitted from the resulting string.

However, direct use of getline is atypical in C++. Instead of reading a line at a time, programmers can use the >> operator for extracting individual pieces of *formatted* data from an input stream. As an example, consider the task of reading a single integer from the user. In Python, we have to first get the raw string and separately compute the integer that those characters represent. Consider, for example, number = **int**(**raw_input**(`'Enter a number from 1 to 10: '`)). In C++, the corresponding code fragment might appear as follows.

```
int number;
cout << "Enter a number from 1 to 10: ";   // prompt without newline
cin >> number;                              // read an integer from the user
```

The >> operator *extracts* data from the stream and stores it in the indicated variable. The static typing of C++ is advantageous in this context. Because number was already designated as an integer, the input characters are automatically converted to the corresponding integer value.

Much as the << operator can be chained in order to insert several pieces of data into an output stream in a single command, it is possible to chain the >> operator can be chained to read in several pieces of data. For example, here is a code fragment that asks the user to enter two numbers and computes their sum.

```
int a, b;
cout << "Enter two integers: ";
cin >> a >> b;
cout << "Their sum is " << a + b << "." << endl;
```

The third line has the effect of inputting two separate integers, the first of which is stored in variable a and the second in b. Formally, the >> operator works by skipping any whitespaces (e.g., spaces or newlines) that reside at the front of the stream, and then interpreting the first non-whitespace token as an integer. The second occurrence of the >> operator on that line causes intermediate whitespace to be removed from the stream and the next token to be interpreted as an integer. Any subsequent characters (including further whitespace) remain on the stream.

When using the stream operator, it does not matter whether the user enters the integers on the same line or different lines, as intermediate newlines and spaces are treated similarly. It also does not matter whether the programmer uses the chained syntax **cin** >> a >> b or two separate commands **cin** >> a followed by **cin** >> b. Both forms are valid, regardless of the user's spacing. Note that this management of input streams is quite different from our usage of the **raw_input** command in Python, which reads a single line. If we expect the user to enter two integers on a single line of input, in Python we have to read the line, then split it into two pieces based on white space, and then attempt to convert each of those pieces to the corresponding integer. A Python version of such a program is given as the solution to Practice 2.31 in the book.

To emphasize the different treatment of whitespace, we reconsider our initial example of querying a person's name. Our original C++ solution used the getline function, to more accurately mirror Python's style. As an alternative, we could write the C++ code as

```
string person;
cout << "What is your name? ";      // prompt the user (without a newline)
cin >> person;                      // input the response
```

Yet this code does not strictly have the same behavior. To highlight the difference, consider the following user session.

```
What is your name? Guido van Rossum
```

After executing the C++ code, the variable person will be assigned the string "Guido", while the subsequent characters (" van Rossum\n") remain on the stream, as it extracted starting with the first non-whitespace character and stopping prior to the next subsequent whitespace. If the use of newlines is significant to the context, a programmer should use a function like getline. However, careful consideration of the whitespace treatment is important when interspersing calls to getline with use of the >> operator. A call to getline removes the ending newline from the stream, but use of the extraction operator reads a token while leaving subsequent whitespace on the stream. Consider the following code fragment.

```
int age;
string food;
cout << "How old are you? ";
cin >> age;
cout << "What would you like to eat? ";
getline(cin, food);
```

A typical user session might proceed as follows.

```
How old are you?  42
What would you like to eat? pepperoni pizza
```

The problem is that after executing the above code, the variable food will be set to the empty string "". The first extraction properly read the age as 42, but the newline character that the user entered after those characters remains on the stream. Even though the user had entered additional input, the call to getline first encounters an apparent empty line because of the remaining newline from the first response. This can be remedied by intentionally reading the blank line before reading the food. A more robust approach relies on use of an ignore function supported by input streams. Another approach for handling line-based input is to always rely on getline to read a single line into a string, and then to use a class named stringstream that supports extracting formatted data from a string; we will discuss the stringstream class in Section 6.6.

Finally, we address the issue of what happens when something goes wrong while attempting to read input. For example, the user might enter the characters hello when an integer was expected. Other unexpected situations may cause failure of the input stream, such as it being closed when a user types *cntr-D* into the console. In Python, such errors result in a formal exception being thrown. For example, a ValueError is reported when trying to get an integer value from an improperly formatted string, and an IOError is reported if the **raw_input** call fails. In C++, an attempt at extracting data from a stream does not throw an exception by default. Instead, the flow of control continues, but with the value of the inputted variable left indeterminate. To support more robust behaviors, the streams have methods that allow a programmer to check various aspects of their state. For example, if a user types non-numeric characters when an integer is expected, the stream's "fail bit" is set to true internally and can be tested with a call to **cin**.fail( ). That bit remains set until explicitly cleared by the programmer. Other state bits can be queried to determine whether the "end of file" has been reached on a stream, or if some other bad state has been reached. Page 182 of our book gives a more robust Python fragment for reading a valid integer from 1 to 10. For comparison, Figure 8 of this document presents a similar code fragment in C++.

6.5 File Streams

C++ supports additional stream classes for managing input and output involving files. Specifically, the <fstream> library defines the ifstream class for reading input from a file, the ofstream class for writing output to a file, and an fstream class that is capable of simultaneously managing input and output for a given file.

We begin with an example using an ifstream instance. If the name of an existing file is known in advance, an input stream can be declared as

```
ifstream mydata("scores.txt");
```

```
number = 0;
while (number < 1 || number > 10) {
  cout << "Enter a number from 1 to 10: ";
  cin >> number;
  if (cin.fail( )) {
    cout << "That is not a valid integer." << endl;
    cin.clear( );                                           // clear the failed state
    cin.ignore(std::numeric_limits<int>::max( ), '\n');     // remove errant characters from line
  } else if (cin.eof( )) {
    cout << "Reached the end of the input stream" << endl;
    cout << "We will choose for you." << endl;
    number = 7;
  } else if (cin.bad( )) {
    cout << "The input stream had fatal failure" << endl;
    cout << "We will choose for you." << endl;
    number = 7;
  } else if (number < 1 || number > 10) {
    cout << "Your number must be from 1 to 10" << endl;
  }
}
```

Figure 8: Robust error-checking with input streams.

If the filename is not known in advance, the stream can be initially declared without a filename and opened as a later operation. The open method accepts the filename as a parameter but, for historical reasons, requires that the name be expressed as a C-style string. Here is an example usage:

```
ifstream mydata;
string filename;
cout << "What file? ";
cin >> filename;
mydata.open(filename.c_str( ));       // parameter to open must be a C−style string
```

The same techniques can be used with an ofstream instance for writing to a file. By default, opening an ofstream causes the target file to be overwritten by a new file, just as with Python's open('scores.txt', 'w'). If you want to append to the end of an existing file, as with Python's open('scores.txt', 'a'), the C++ command is

```
ofstream datastream("scores.txt", ios::app);
```

The more general fstream class can be used to simultaneously manage input and output from the same file, although coordinating such manipulations takes more care.

6.6 String Streams

We have seen how **cin** and **cout** manage the console and how the file streams are used to manage files. All of the stream operators offer convenient support for reading or writing formatted data. As an example, if we have an integer age, the command **cout** << age converts the integer into the corresponding characters used to display that number (e.g., "42"). But what if we want to compute the string representation of an integer, not for immediate output, but perhaps to save in a **string** variable? In Python, the syntax displayedAge = **str**(age) produces such a string.

We avoided this issue at the end of Section 4, because the conversion is not quite as direct in C++. Instead, we can use the stringstream class, included from the <sstream> library. This class allows us to use the stream operators to insert formatted data into a string or to extract formatted data from that string. For example, here is code that produces a string based upon an integer value.

```
int age(42);
string displayedAge;
stringstream ss;
ss << age;           // insert the integer representation into the stream
ss >> displayedAge;  // extract the resulting string from the stream
```

String streams can also be used to convert in the other direction, starting with a string and parsing it to extract pieces of data. For example, in Section 6.4 we discussed differences between tokenized input in C++ versus Python's style of using **raw_input** to read a line and then subsequently splitting the tokens on that line. In C++, we can emulate such a style by using getline to read a single line as a string, and then using a stringstream to manage subsequent extractions of formatted data.

7 Classes in C++

Classes provide the same abstraction for storing and manipulating information in C++ as they do in Python. In Section 7.1 we discuss classes from the perspective of a user of a class, with rather similar syntax between Python and C++, other than the need for a clear type declaration in C++. In Section 7.2 we introduce the syntax for *defining* classes in C++. We discuss the additional complexities that arise when defining a class in C++ due to the additional burden of type declarations and other such specifications that assist in more rigorous compile-time checking.

7.1 Using Instances of a Class

Thus far, most of our examples have used primitive data types (e.g., **int**, **double**). However, **string** is a class type, as are the various forms of streams from the previous section. In C++, when a variable of a class type is declared, an instance of that class is automatically constructed. Two examples of such a declaration are the following:

```
string s;                   // relies on default constructor without parameters
string greeting("Hello");   // explicit parameter sent to the constructor
```

In the first declaration, a new string instance is created using the *default constructor* of the class, that is, a constructor that accepts zero parameters. In the case of strings, the default constructor produces the empty string. The second syntax invokes a form of the **string** class constructor that accepts a parameter to designate the initial contents of the new string.

As an aside, we wish to warn against a few potential mistakes in the declaration syntax. First, when relying upon the default constructor, we did *not* use empty parentheses. Had we used parentheses, it would have looked like the following.

```
string s( );     // WARNING!!!!! A function declaration
```

Although the intent may be to send zero parameters to the constructor, the C++ parser deems this syntax as the signature for a *function* named s, taking zero parameters and returning a **string** (that is, after all, how such a function's signature would appear). As a second example, we wish to emphasize that when parameters are specified in a declaration, they are included after the *variable* name rather than after the *class* name. Compare the following illegal syntax with the original correct syntax.

```
string("Hello") greeting;        // parameters in wrong place
```

With that said, there is a scenario in which the parameters to a constructor appear after the class name. This happens when a programmer wishes to construct an *unnamed* instance as part of a larger expression, such as **cout** << **string**(20, '-'). We will see another such use in the coming section involving the robust version of a Point class. A line of that code will appear as follows.

```
double mag = distance( Point( ) );      // measure distance to the origin
```

The syntax Point( ) in this expression causes the construction of a new Point instance based on the default constructor and in this context, the explicit parentheses are necessary.

Once an instance of a class has been constructed, C++ uses typical object-oriented syntax such as greeting.replace(1, 4, "ey") to call the replace method on the string identified as greeting.

7.2 Defining a Class

To demonstrate the syntax for defining a C++ class, we revisit several Python examples from Chapter 6 of our book. We begin with the simple version of the Point class as given in Figure 6.3 on page 206. The corresponding C++ version of that class is given in Figure 9 of this document. There are several important aspects to discuss in comparing C++'s syntax to Python's.

Explicit declaration of data members

The issue of static typing arises prominently in a class definition as all data members must be explicitly declared. Recall that in Python, attributes of a class were simply introduced by assignment statements within the body of the constructor. In our C++ example, we explicitly declare the two data members at lines 3 and 4.

Constructor

Line 7 of our code is the constructor, although the syntax requires some explanation. The line begins with the name of the class itself (i.e., Point) followed by parentheses. The constructor is a function, with this particular example accepting zero parameters. However, unlike other functions, there is no designated return value in the signature (not even **void**).

The next piece of syntax is the colon followed by _x(0), _y(0). This is what is known as an ***initializer list*** in C++. It is the preferred way to establish initial values for the attributes (we are not allowed to express initial values on lines 3 and 4). Finally, we see the syntax { }. This is technically the body of the constructor. Some classes use the constructor body to perform more intricate initializations. In this case, having already initialized the two variables, there is nothing else for us to do. So the { } serves syntactically as a placeholder for the function body (somewhat like **pass** in Python).

```
 1  class Point {
 2  private:
 3    double _x;                           // explicit declaration of data members
 4    double _y;
 5
 6  public:
 7    Point( ) : _x(0), _y(0) { }          // constructor
 8
 9    double getX( ) const {               // accessor
10      return _x;
11    }
12
13    void setX(double val) {              // mutator
14      _x = val;
15    }
16
17    double getY( ) const {               // accessor
18      return _y;
19    }
20
21    void setY(double val) {              // mutator
22      _y = val;
23    }
24
25  };                                       // end of Point class (semicolon is required)
```

Figure 9: Implementation of a simple Point class.

Implicit self-reference

A careful reader will have already noticed another major distinction between the class definition in C++ and the same class in Python. The **self** reference does not appear as a formal parameter nor is it used when accessing members of the instance. Remember that we have explicitly declared _x and _y to be attributes of a point. Because of this, the compiler recognizes those identifiers when used within the body of our methods (for example at line 10). For those who miss the self-reference, it is implicitly available in C++ yet with the name **this**. It can be useful, for example, when passing the object as a parameter to an outside function. Technically, **this** is a *pointer* variable (a concept that will be introduced in Section 8.3).

Access control

Another distinction in C++ is the use of the terms **public** and **private** within the class definition. These relate to the issue of ***encapsulation***. With Python, we addressed this issue in Chapter 7.6 of our book, differentiating at the time between what we considered "public" versus "private" aspects of a class design. Public aspects are those that we expect other programmers to rely upon, while private ones are considered to be internal implementation details that are subject to change. Yet Python does not strictly enforce this designation. Instead, we relied upon naming conventions, using identifiers that start with an underscore (e.g., _x) to infer privacy.

In C++, these designators serve to declare the desired ***access control*** for the various members (both data members and functions). The use of the term **private** at line 2 affects the declarations at lines 3 and 4, while the term **public** at line 6 effects the subsequent declarations. The compiler enforces these designations within the rest of the project, ensuring that the private members are not directly accessed by any code other than our class definition.

Designating accessors versus mutators

In our Python book, we used the notion of an ***accessor*** as a method that cannot alter the state of an object, and a ***mutator*** as a method that might alter the state. This distinction is formalized in C++ by explicitly placing the keyword **const** for accessors at the end of the function signature but before the body. In our example, we see this term used in the signature of getX at line 9 and again for getY at line 17. We intentionally omit such a declaration for the mutators setX and setY.

As with access control, these **const** declarations are subsequently enforced by the compiler. If we declare a method as **const** yet then try to take an action that risks altering any of the attributes, this causes a compile-time error. Furthermore, if a caller has an object that had been designated as immutable, the only methods that can be invoked upon that object are ones that come with the **const** guarantee.

A robust Point class

To present some additional lessons about class definitions in C++, we provide a more robust implementation of a Point class modeled upon the Python version from Figure 6.4 on page 213 of our book. Our C++ version is shown in Figures 10 and 11 of this document.

Our first lesson involves the constructor. In Python, we declared a constructor with the signature **def** __init__(**self**, initialX=0, initialY=0). This provided flexibility, allowing a caller to set initial coordinates for the point if desired, but to use the origin as a default. The C++ version of this constructor is given at line 7.

In our first version of the class, we emphasized that data members of the class can be accessed without explicitly using a self-reference, for example using a syntax like _x rather than Python's **self**._x. The same convention is used when the body of one member function invokes another. For example, our implementation of the normalize method relies on a call to the distance method at line 26 and the scale method at line 28. Those calls are implicitly invoked on the same Point instance upon which normalize has been invoked. In contrast, the use of the constructor Point( ) at line 26 instantiates a new (default) point as a parameter to the distance function.

Lines 31–33 are used to support the syntax p + q for the addition of two points. This behavior is akin to the __add__ method in Python, yet in C++ the semantics are defined using **operator**+ as the "name" of the method. In this context, the *left-hand operand* p serves as the implicit instance upon which the **operator**+ method is invoked, while q appears as an explicit parameter in the signature. The **const** declaration that we make at line 31 designates that the state of p is unaffected by the behavior (q is unaffected as well, but we delay discussion of that issue until Section 8.2).

Lines 35–41 support two different notions of multiplication: multiplying a given point by a numeric constant, and computing the dot product of two points. Our original Python implementation accomplished this with a single function definition that accepted one parameter. Internally it performed dynamic type-checking of that parameter and determined the appropriate behavior depending on whether the second operand was a point or a number. In C++, we provide two different implementations. The first accepts a **double** and returns a new Point; the second accepts

```
class Point {
private:
  double _x;
  double _y;

public:
  Point(double initialX=0.0, double initialY=0.0) : _x(initialX), _y(initialY) { }

  double getX( ) const { return _x; }       // same as simple Point class
  void setX(double val) { _x = val; }       // same as simple Point class
  double getY( ) const { return _y; }       // same as simple Point class
  void setY(double val) { _y = val; }       // same as simple Point class

  void scale(double factor) {
    _x *= factor;
    _y *= factor;
  }

  double distance(Point other) const {
    double dx = _x - other._x;
    double dy = _y - other._y;
    return sqrt(dx * dx + dy * dy);          // sqrt imported from cmath library
  }

  void normalize( ) {
    double mag = distance( Point( ) );       // measure distance to the origin
    if (mag > 0)
      scale(1/mag);
  }

  Point operator+(Point other) const {
    return Point(_x + other._x, _y + other._y);
  }

  Point operator*(double factor) const {
    return Point(_x * factor, _y * factor);
  }

  double operator*(Point other) const {
    return _x * other._x + _y * other._y;
  }
};    // end of Point class (semicolon is required)
```

Figure 10: Implementation of a robust Point class.

```
43 // Free-standing operator definitions, outside the formal Point class definition
44 Point operator*(double factor, Point p) {
45   return p * factor;                          // invoke existing form with Point as left operand
46 }
47
48 ostream& operator<<(ostream& out, Point p) {
49   out << "<" << p.getX( ) << "," << p.getY( ) << ">";      // display using form <x,y>
50   return out;
51 }
```

Figure 11: Supplemental operator definitions involving Point instances.

a Point and (coincidentally) returns a **double**. Providing two separate declarations of a method is termed ***overloading*** the signature. Since all data is explicitly typed, C++ can determine which of the two forms to invoke at compile-time, based on the actual parameters.

Line 42 ends our formal Point class declaration. However, we provide two supporting definitions following that line. The first of those is used to support a syntax such as 3.25 * p. The earlier definition of **operator*** from lines 35–37 supports the * operator when a Point instance is the *left-hand* operand (e.g., p * 3.25). C++ does not allow the class definition of the right-hand operand to directly impact the behavior. Yet if the left-hand operand's type does not provide an adequate definition (as with type **double** in the expression 3.25 * p), C++ looks for a free-standing **operator*** function with a matching signature. So at lines 44–46, we provide a definition for how * should behave when the first operand is a **double** and the second is a Point. Notice that both operands appear as formal parameters in this signature since we are no longer within the context of a class definition. The body of our method uses the same simple trick as in our Python implementation, commuting the order so that the point becomes the left-hand operand (thereby, invoking our previously defined version). As an aside, notice that we did not have to have such an additional definition for **operator**+ since both operands (and thus the left-hand one) are Point instances for addition.

Finally, lines 48–51 are used to produce a text representation of a point when inserted into an output stream. A typical syntax for such a behavior is **cout** << p. Again, we define this behavior outside of the context of the Point class because the left-hand operand is the output stream, and because we are not the authors of the ostream class. In our implementation, line 49 inserts our desired output representation into the given output stream. We use the formal parameter out rather than **cout** so that a user can apply this behavior to any output stream instance. The declared return type on line 48 and the return statement at line 50 are technically required to allow for multiple << operations to be chained in a single expression. For example, the syntax **cout** << p << " is good" is evaluated as (**cout** << p) << " is good", with the result of the first evaluation being an output stream used in the second operation. The use of the & symbol twice on line 48 (for both the return type and the first parameter type) is a technicality that we will address in Section 8.2.

7.3 Inheritance

In Chapter 9 of our book, we provided several examples of the use of inheritance in Python. We will show two of those examples, translated to C++. First we define a DeluxeTV class modeled closely after the version in Figure 9.2 of the book which used a SortedSet. Although we omit the

```
 1 class DeluxeTV : public Television {
 2   protected:
 3     set<int> _favorites;
 4
 5   public:
 6     DeluxeTV( ) :
 7       Television( ),      // parent constructor
 8       _favorites( )       // empty set by default
 9     { }
10
11     void addToFavorites( ) { if (_powerOn) _favorites.insert(_channel); }
12
13     void removeFromFavorites( ) { if (_powerOn) _favorites.erase(_channel); }
14
15     int jumpToFavorite( ) {
16       if (_powerOn && _favorites.size( ) > 0) {
17         set<int>::iterator result = _favorites.upper_bound(_channel);
18         if (result == _favorites.end( ))
19           result = _favorites.begin( );       // wrap around to smallest channel
20         setChannel(*result);
21       }
22       return _channel;
23     }
24 };   // end of DeluxeTV
```

Figure 12: Implementing a DeluxeTV class through inheritance.

presumed definition for a basic Television class, our complete code for the DeluxeTV class is given in Figure 12. The use of inheritance is originally indicated at line 1 by following the declaration of the new class with a colon and then the expression **public** Television. With that designation[2], our DeluxeTV class immediate inherits all attributes (e.g., powerOn, channel) and all methods (e.g., setChannel) from the parent. What remains is for us to define additional attributes or to provide new or updated implementations for methods that we want supported.

At line 3, we declare a new attribute to manage the set[3] of favorite channel numbers. We wish to draw particular attention to the use of the word **protected** at line 2. Until now, we have used two forms of access control: **public** and **private**. Members that are public can be accessed by code outside of the class definition, while members that are private can only be accessed from within the original class definition. The purpose of privacy is to encapsulate internal implementation details that should not be relied upon by others. Yet with the use of inheritance, there is need for a third level of access. When one class inherits from another, the question arises as to whether code for the child class should have access to members inherited from the parent. This is determined by the access control designated by the parent. A child class cannot directly access any members declared as **private** by the parent. However, the child is granted access to members designated as **protected** by the parent.

[2]For the sake of simplicity, we will not discuss the precise significance of the term **public** on line 1.
[3]We will discuss the **set** class and other containers in Section 10.

In this particular setting, the important point is not actually our use of **protected** at line 2. What matters to us is how the original attributes of the Television class were defined. For our DeluxeTV code to work, it must be that television attributes were originally declared as

```
protected:
  bool _powerOn;
  int _channel;
  ...
```

If those had been declared as **private**, we would not have the necessary access to implement our DeluxeTV. The original designer of the television may not have known that we would come along and want to inherit from it, but an experienced C++ programmer will consider this possibility when designing a class. In our DeluxeTV definition, the declaration of attribute favorites as **protected** is not for our own benefit, but to leave open the possibility that someone else may one day want to design a SuperDeluxeTV that improves upon our model. As an alternative to protected data, a parent can provide protected member functions to encapsulate the private state.

The second aspect of our example we wish to discuss is the definition of our constructor, at lines 6–9. In our Python version, the new constructor begins with an explicit call to the parent constructor, using the syntax, Television.__init__(**self**). That was used to establish the default settings for all of the inherited attributes. In C++, we can invoke the parent constructor as part of the initializer list using the syntax Television( ) at line 7. This calls the parent constructor without sending any explicit parameters. To be honest, in this particular example, line 7 is superfluous. If we do not explicitly call the parent constructor, C++ will do so implicitly. However, an explicit call is necessary when parameters are to be sent to the parent constructor (as in our second example). In this example, our default initialization of favorites at line 8 is also superfluous.

Lines 11–23 of our DeluxeTV code provides three new behaviors. The precise details of those methods depend on knowledge of the **set** class; as such, we will revisit some of this code in Section 10. Our purpose for the moment is to demonstrate the use of inheritance. We draw attention to the fact that we are able to access the inherited attributes, _powerOn and _channel, as well as our new attribute _favorites when implementing the methods. We also call the inherited method setChannel.

A Square class

As a second example of inheritance, Figure 13 provides a C++ rendition of our original Square class from Chapter 9.4.2 of our book. The Square inherits from a presumed Rectangle class. We do not introduce any new attributes for this class, so our only responsibility for the constructor is to ensure that the inherited attributes are properly initialized. To this end, we invoke the parent constructor at line 4. In this case, we need the explicit call in order to pass the appropriate dimensions and center. Had we not done so, an implicit call would have been made to the *default* version of the rectangle constructor, leading to incorrect semantics for our square.

The remainder of the definition is meant to provide new getSize and setSize methods, while also overriding the existing setHeight and setWidth methods so that a change to either dimension affects both. We use the same approach as our Python version. We override the existing methods at lines 7 and 8, changing their behaviors to call our new setSize method. Our setSize method then relies upon the *parent* versions of the overridden setWidth and setHeight methods to enact the individual changes to those values. The expression Rectangle:: before the method names at lines 11 and 12 is a ***scope resolution***, indicating our desire to invoke the definitions of those behaviors from the parent Rectangle class, rather than the corresponding methods of the Square class.

```
class Square : public Rectangle {
public:
  Square(double size=10, Point center=Point( )) :
    Rectangle(size, size, center)        // parent constructor
  { }

  void setHeight(double h) { setSize(h); }
  void setWidth(double w) { setSize(w); }

  void setSize(double size) {
    Rectangle::setWidth(size);       // make sure to invoke PARENT version
    Rectangle::setHeight(size);      // make sure to invoke PARENT version
  }

  double getSize( ) const { return getWidth( ); }
};  // end of Square
```

Figure 13: Implementing a Square class based upon a Rectangle.

8 Object Models and Memory Management

Python supports a consistent model in which all identifiers are inherently references to underlying objects. Use of the assignment operator as in a = b causes identifier a to be reassigned to the same underlying object referenced by identifier b. These semantics are consistently applied to all types of objects. The assignment semantics also apply to the passing of information to and from a function, as described in Chapter 10.3.1 of our book. Upon invocation, the formal parameters are assigned respectively to the actual parameters indicated by a caller. The return value is communicated in a similar fashion. As a simple example, assume that we define the following Python function for determining whether a given point is equivalent to the origin:

```
def isOrigin(pt):
  return pt.getX( ) == 0 and pt.getY( ) == 0
```

Now assume that the caller invokes this function as isOrigin(bldg), where bldg is an identifier that references a Point instance. Figure 14 diagrams the underlying configuration. This scenario is the precise result of the system performing an implicit assignment pt = bldg, setting the formal parameter to the actual parameter.

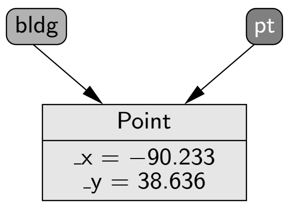


Figure 14: An example of parameter passing in Python.

C++ provides more fine-tuned control than Python, allowing the programmer a choice between three different semantic models for storing and passing information. In this section, we examine the correspondence between identifiers and underlying values in C++.

8.1 Value Variables

The most commonly used model in C++ is that of a ***value variable***. Declarations such as

```
Point a;
Point b(5,7);
```

cause the system to reserve memory for storing newly constructed points. Because all data members for a Point are explicitly declared in the class definition, the system can determine precisely how much memory is required for each instance. The translation from a name to a particular instance is handled at *compile-time*, providing greater run-time efficiency than Python's run-time mapping.

To portray the semantics of a value variable, we prefer a diagram in the style of Figure 15, without any independent concept of a reference. The assignment semantics for a value variable is very different from Python's. The command a = b assigns Point a the *value* currently held by Point b, as diagrammed in Figure 16. Notice that names a and b still represent two distinct points.

The semantics of value variables is manifested as well in the passing of information to and from a function. Consider the following C++ analog to our earlier Python example.

```
bool isOrigin(Point pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

When a caller invokes the function as isOrigin(bldg), the formal parameter Point pt is implicitly initialized as if using the copy constructor syntax,

```
Point pt(bldg);
```

Note that the formal parameter pt does not become an alias for the actual parameter. It is a newly allocated Point instance with state initialized to match that of the actual parameter bldg. Figure 17 portrays this scenario. As a result, changes made to the parameter from within the function body have no lasting effect on the caller's object. This style of parameter passing is generally termed ***pass-by-value***, as originally discussed on page 351 of our book.

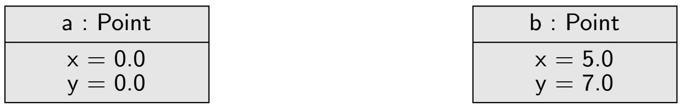


Figure 15: The declaration of two separate value variables.

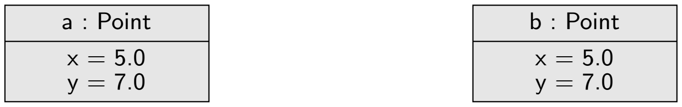


Figure 16: The effect of an assignment a = b upon value variables.

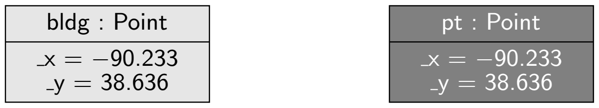


Figure 17: An example of passing by value in C++.

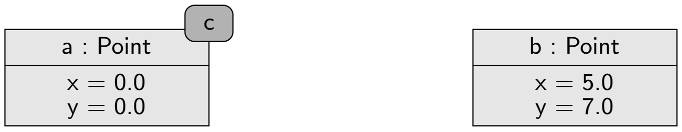


Figure 18: The name c is an example of a reference variable in C++.

8.2 Reference Variables

A second model for a C++ variable is commonly termed a ***reference variable***. It is declared as

```
Point& c(a);   // reference variable
```

Syntactically, the distinguishing feature is the use of the ampersand. This designates c as a new name, but it is not a new point. Instead, it becomes an alias for the existing point, a. We choose to diagram such a situation as in Figure 18.

This is closer to the spirit of Python's model, but still not quite the same. A C++ reference variable must be bound to an existing instance upon declaration. It cannot be a reference to nothing (as is possible in Python with the None value). Furthermore, the reference variable's binding is static in C++; once declared, that name can no longer be re-associated with some other object. The name c becomes a true alias for the name a. The assignment c = b does not rebind the name c; this changes the *value* of c (also known as a).

Reference variables are rarely used as demonstrated above, because there is little need in a local context for a second name for the same object. Yet the reference variable semantics becomes extremely important in the context of functions. We can use a ***pass-by-reference*** semantics by using the ampersand in the declaration of a formal parameter, as in the following revision of isOrigin.

```
bool isOrigin(Point& pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

This leads to a model similar to Python in that the formal parameter becomes an *alias* for the actual parameter. There are several potential advantages of this style. For larger objects, passing the memory address is more efficient than creating and passing a copy of the object's value. Passing by reference also allows a function to intentionally manipulate the caller's object. If we do not want to allow a function to mutate its parameter, yet we want the efficiency of passing it by reference, a **const** modifier can be declared with the formal parameter, as in

```
bool isOrigin(const Point& pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

With such a signature, the point will be passed by reference but the function promises that it will in no way modify that point (a promise that is enforced by the compiler).

Given this discussion of reference variables, we revisit our earlier definition of the robust Point class from Section 7.2. Notice that our original signature for the distance method on line 19 of Figure 10 appeared as

```
double distance(Point other) const {
```

The point serving as the parameter is being passed "by value," causing a local copy to be made. As a value parameter, we do not bother to make a **const** declaration because it does not matter whether the function body modifies the local value (although it does not). Since a point has several fields, it may be more efficient to pass it by reference. In that case, we will clearly designate it as a constant reference with the following revised signature.

```
  double distance(const Point& other) const {
```

The **const** for the parameter declaration assures the caller that the the function will not modify its value, while the **const** declaration that follows the signature designates the method as an accessor, meaning that the point upon which it is invoked is unchanged.

We might similarly revise the **operator**+ method to receive its parameter as a constant reference.

```
Point operator+(const Point& other) const {
  return Point(_x + other._x, _y + other._y);
}
```

Note that the return type remains declared as a value rather than a reference. This is because our local return value is a transient object that will soon be destroyed. It would be unsafe to return a reference to such an object to the caller, so we must send its value.

As a final example, our original version of the **operator**<< method from Figure 11 appeared as

```
ostream& operator<<(ostream& out, Point p) {
  out << "<" << p.getX( ) << "," << p.getY( ) << ">";      // display using form <x,y>
  return out;
}
```

While we could revise this definition to send the Point as a constant reference, we wish to draw attention to the treatment of the ostream as both a parameter and return value. Notice that the out parameter is designated as a reference. This is because we wish to insert data into the actual stream indicated by the caller, not a "copy" of the stream (in fact, streams cannot legally be copied). Note as well that this parameter is not designated as a **const** reference, because our insertion of data impacts the state of that stream.

Because streams cannot be copied, we must return it as a reference using ostream& as the declared return type. In our earlier discussion of **operator**+ we emphasized that it was not safe to return its resulting point as a reference. The problem there was that the result had been created in the local scope of the function. In the case of **operator**<<, the stream that we are returning originated with the caller. Therefore, we may safely return a reference to it knowing that it will continue to exist after our function completes.

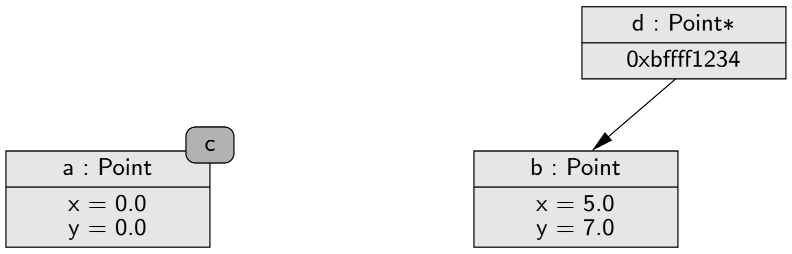


Figure 19: Variable d is an example of a pointer whose value is the address of instance b.

8.3 Pointer Variables

C++ supports a third model for variables known as a ***pointer***. This has semantics closest to Python's model, but the syntax is quite different. A C++ pointer variable is declared as follows:

```
Point *d;      // d is a pointer variable
```

The asterisk in this context declares that d is not a Point itself, but a variable that can store the memory address of a Point. Pointers are more general than reference variables in that a pointer is allowed to point to nothing (using the keyword NULL in C++) and a pointer can be dynamically reassigned to the address of another instance. A typical assignment statement is as follows:

```
d = &b;
```

This leads to a configuration diagrammed in Figure 19. We intentionally portray d as a separate entity because it is itself a variable stored in memory and manipulated, whose value just happens to be a memory address of some other object. In order to manipulate the underlying Point with this variable, we must explicitly dereference it. While the syntax d represents a pointer, the syntax *d represents the thing it addresses (as a mnemonic, consider the original declaration Point *d which suggests that *d is a Point). For example, we could call the method (*d).getY( ), which returns the value 7.0 in this case. The parentheses are necessary due to operator precedence. Because this syntax is bulky, a more convenient operator −> is supported, with equivalent syntax d−>getY( ).

Pointers provide several additional opportunities for a C++ programmer. A pointer can be sent as parameter to a function, as demonstrated in the following revision of isOrigin.

```
bool isOrigin(Point *pt) {
  return pt−>getX( ) == 0 && pt−>getY( ) == 0;
}
```

Technically, what happens is that we declare a new local pointer, with the value of that pointer set to the value of the pointer sent by the caller. As a result, the function body has indirect access to the same underlying object that the pointer addresses. This provides similar opportunity to when parameters are passed by value, but it allows the additional possibility of the caller sending a NULL pointer (recall that reference variables must be bound to something). The following section introduces another use of pointers in C++, that of managing dynamically-allocated objects.

8.4 Dynamic Memory Management

With value variables, C++ handles all issues of memory management. When a declaration is made, such as Point a, the system reserves memory for storing the state of the object. Furthermore, when that variable declaration goes out of scope (for example, if a local variable within a function body), the system automatically destroys the object and reclaims the memory for other purposes. Generally, this automatic memory management eases the burden upon the computer.

However, there are circumstances when a programmer wants to take a more active role in controlling the underlying memory management. For example, we might want a function that creates one or more objects that are to remain in memory beyond the context of the function. In C++, such a dynamic instantiation is indicated using the keyword **new** in a context such as **new** Point( ) for a default construction, or **new** Point(5,7) for a non-default construction. Formally, the **new** operator returns the memory address at which the constructed object is stored. To be able to further interact with the object, we must be able to locate it. A common approach is to use a pointer variable to remember the location.

```
Point *p;               // declare pointer variable (not yet initialized)
p = new Point( );       // dynamically allocate a new Point instance, storing its address
```

With this code fragment, two different pieces of memory are being used. A certain number of bits are set aside for managing the pointer variable p, while another set of bits are set aside for storing the state of the underlying Point instance. The key is that with the dynamic allocation of the point, that instance will remain in memory even when the variable p goes out of scope (causing the reclamation of the variable p but not the object to which it points).

Management of dynamically-allocated objects requires more care. If a program were to lose track of the memory location of the object (such as by reassigning a pointer variable to a different location), the original object would remain in memory yet be inaccessible[4]. Such a mistake is known as a ***memory leak*** and a program that continues to allocate such objects while never deallocating them consumes more and more of the computer's memory as it runs. In C++, the programmer has the burden of explicitly destroying a dynamically-allocated object when it is no longer needed. This is done by using a syntax such as **delete** p in the above example. The expression after the keyword **delete** specifies the address of the object to be deleted. However, **delete** must only be used on objects that were dynamically-allocated. If you were to specify the address of a standard value variable an error occurs when the system subsequently attempts the automatic deallocation.

8.5 Treatment of Arrays

As a holdover from C, arrays are treated rather unusually, especially when compared to the more object-oriented container classes that we will discuss in Section 10. In Sections **??** and 4.2 we had introduced the basic declaration and use of arrays in C++. We considered the following example.

```
double measurements[300];
```

This declaration causes the system to create an array of 300 double values. A particular entry of that array can be accessed with an expression such as measurements[7]. In this section, we wish to discuss the treatment of the array as a whole.

[4]In contrast, Python detects inaccessible objects and reclaims them, as discussed in Chapter 10.1.3 of the book. However, Python must perform additional bookkeeping for all objects to perform that task, thus trading efficiency for convenience.

The expression measurements, without any explicit indexing, is a legal syntax in C++. Intuitively, that expression relates to the array as a whole, but the semantics is not the same as with a primitive type. The expression measurements represents the *memory address of the beginning of the array.* In fact, the formal data type for this expression is a **double**∗ (i.e., a pointer to a double).

The significance of this semantics can be demonstrated with an example. Assume that our previous declared measurements has been populated with data. We might wish to create a second array and to copy the original data to the secondary array. The following is an errant attempt.

```
double backup[300];
backup = measurements;     // does not actually copy the underlying array
```

The first line is a legitimate declaration of a second array, able to store 300 double values. The problem is that the assignment backup = measurements changes the pointer backup to have a value equal to the address currently known as measurements. In affect, we have made the variable backup an alias for the original array (while leaving the newly allocated array inaccessible).

There is no direct way to make a copy of an entire array. Instead, we could write our own loop to initialize the second array with the same values stored in the original.

```
double backup[300];
for (int i=0; i < 300; i++)
  backup[i] = measurements[i];
```

The array variable can be used directly as a pointer. Since measurements is a pointer to the first array location, the expression ∗measurements represents the double stored at that location (i.e., measurements[0]). C++ pointers support a notion of arithmetic in that the expression measurements + 7 represents the address that is seven entries beyond the start of the array. Thus ∗(measurements + 7) is equivalent in meaning to measurements[7].

We should also note that C++ does not make any effort to ensure that our index is legitimate. For example, given our earlier declaration, use of the expression measurements[350] is legal, but the affect is likely disastrous. Given knowledge of the starting address for the array, this expression refers to the "double" that is located 350 entries away from the start of the array. However, since our array was declared with only 300 entries, the memory location that is 350 steps away from the beginning is not part of the array. It could vary well be bits that are being used to represent some other object. In comparing this treatment to that of Python's list, we note the trade-off between efficiency and convenience. If we were to attempt to access measurements[350] in Python for a list that did not actually have that many entries, an IndexError is thrown. Python checks the validity of the index at run-time, yet this check requires a few extra steps internally. C++ opts for greater efficiency by blindly going to the indicated memory location, assuming that the programmer has designated a valid index.

The treatment of arrays in C++ also impacts the way in which we send an array as a parameter to a function. For example, we might wish to support a function sum that would support a natural syntax such as total = sum(measurements). Unfortunately, such a calling signature is nontrivial. The challenge relates to the fact that arrays cannot be directly copied en masse. When sending measurements as a parameter, what is really being sent is the pointer value. Yet within the context of the function, we need to know not only where the array begins in memory, but also how many entries there are. So a common approach to implementing such a function involves two separate parameters, as demonstrated in the following example.

```
double sum(double data[ ], int n) {
  double temp = 0.0;
  for (int i=0; i < n; i++)
    temp += data[i];
  return temp;
}
```

The first parameter designates that data is in effect an array of doubles, although all that is sent is a pointer to the beginning of the array. The second parameter is used to denote the array's length. The caller must use a syntax such as total = sum(measurements, 300) when invoking our function. Inside the body of the function, we index the array using a standard syntax such as data[i]. Because the first parameter is really just a pointer, some authors will use the equivalent signature

```
double sum(double *data, int n) {
```

Although it is not syntactically clear that data points to an array of doubles rather than a single double, documentation for such a function would designate the proper usage.

We also note that either forms of our sum function can be used to compute the sum of a desired subarray, simply by designating the "start" of the subarray as the first parameter and the length of the subarray as the second. For example, the call sum(&measurements[50], 10) will compute the equivalent of Python's sum(measurements[50:60]). It involves the subarray starting at the *address* of measurements[50], containing ten entries. Fans of pointer arithmetic could make the same call using the syntax sum(measurements + 50, 10).

Finally, we wish to note that arrays can be dynamically-allocated rather than relying on automatic memory management. This is particular useful when we need to create an array, but we do not know the proper size of the array until run-time. In that case we can begin by declaring our "array" as a pointer to the base type, such as

```
double *measurements;
```

When we wish to dynamically allocate an array of a particular size, we can do so using a syntax,

```
measurements = new double[numEntries];     // assuming numEntires is well-defined
```

This dynamically allocates an array of **double** values, returning the address of the first entry. As was the case in Section 8.4, a programmer who dynamically allocates an array is ultimately responsible for releasing the memory when it is no longer needed. However, to deallocate an array, a programmer must use the **delete[ ]** operator rather than **delete**. Thus if measurements were dynamically allocated as above, it can be destroyed with the command **delete**[ ] measurements.

8.6 Constructors, Destructors, and the Assignment Operator

Constructors in C++ serve a similar purpose as they do in Python. They are primarily responsible for initializing the state of a newly created object. For example, the simple Point class of Figure 9 supports a constructor that initializes both data members to the value 0.0. Such a zero-parameter form is known as a ***default constructor*** and is invoked with a user syntax such as

```
Point a;
```

The constructor for the more robust Point class of Figure 10 supports additional calling signatures for the user's convenience. It could be invoked for example as

```
Point b(5,7);
```

The Point class also supports another constructor form known as a ***copy constructor***, even though we did not provide any explicit code as such. This allows a new instance to be initialized based upon the value of an existing instance. For example, given our earlier definition of point b, we could instantiate a new point as follows.

```
Point c(b);                   // Existing point b is the parameter
```

This creates and initializes a new instance c, which is given an initial value patterned upon instance b. But it is important to understand that the new instance is independent, in the sense that it has its own memory and that subsequent changes to one of these points will not affect the other. The first is simply used as a model for the second, as the new point's _x value is set to the same as the original's _x, and the new _y is initialized to the value of the original _y. If we were to describe the behavior of the copy constructor it would be coded as follows.

```
    Point(const Point& other) : _x(other._x), _y(other._y) { }          // copy constructor
```

Yet we did not explicitly need such a definition in our Point class. C++ provides an implicit copy constructor for every class that does not explicitly define one. By default, it performs a member-by-member copy. The primary reason for a default copy constructor is that the system needs to do its own copying when passing value parameters. Recall that when a parameter is passed by value, the formal parameter is a local instance that is initialized to match the caller's actual parameter. The copy constructor is the mechanism used to create that local instance.

However, some classes need a more specialized behavior to provide correct copying semantics. In particular, there is a potential ambiguity for a class that contains a pointer as a data member. That pointer might be addressing some memory that was allocated specifically for the state of the given instance, or it might be pointing to memory that inherently "belongs" to some other object. In the former case, the correct semantics is to do a deep copy of the object being referenced. In the latter case, the correct semantics is likely to replicate the pointer but not to replicate the underlying object. The member-by-member copying mechanism of the default copy constructor in effect is a ***shallow copy***, creating a pointer whose value is the same address as the pointer that is being copied[5]. Yet C++ allows the designer of a class to override the semantics of the copy constructor.

As a motivating example, we develop a simple version of a TallySheet class for integer values, akin to that used in Chapter 8.5.3 of our book for computing frequency counts. We wish to have an array of counters for the various numbers that are added to the data set, yet we do not know how big of an array to declare until we are told the range of values that may occur. For that reason, we must begin by declaring an **int**∗ data member, and then later set that pointer to the address of a properly allocated array of integers. The beginning of our class appears in Figure 20. The pointer _tallies is declared at line 6. It is not until the primary constructor is called before we can compute the required size of the array and then dynamically allocate it (line 10). The complication arises if someone were to make a copy of a tally sheet. For example, we might imagine a user who wants to make a copy of a partial tally, and then do further simulation on the copy while ensuring that the original version of the counts remains unchanged.

[5]We strongly recommend that you revisit Chapter 10.2 of our book for a discussion of these issues in Python.

```
 1  class TallySheet {
 2    private:
 3      int _minV;
 4      int _maxV;
 5      int _size;
 6      int* _tallies;
 7
 8    public:
 9      TallySheet(int minVal, int maxVal) :
10               _minV(minVal), _maxV(maxVal), _size(maxVal-minVal+1), _tallies(new int[_size]) {
11        for (int i=0; i < _size; i++)
12          _tallies[i] = 0;
13      }
```

Figure 20: The beginning of a simple TallySheet class.

It would be a mistake for us to rely on the system-provided copy constructor. Because the _tallies member is a pointer, the default copy would result in a new tally sheet with its _tallies pointer set to the same *address* as the original _tallies value. In effect, the two TallySheet instances would be sharing the same underlying array, thereby interfering with an accurate count. The proper semantics for copying a tally sheet is to create a deep copy, giving the new instance its own array of counts, albeit initialized based upon the existing counts at that time. We provide a specialized version of the copy constructor by defining an additional constructor that accepts another TallySheet instance as a parameter. For our class, we implement the proper copying semantics as follows.

```
14      TallySheet(const TallySheet& other) :
15               _minV(other._minV), _maxV(other._maxV), _size(other._size), _tallies(new int[_size]) {
16        for (int i=0; i < _size; i++)
17          _tallies[i] = other._tallies[i];
18      }
```

The new sheet gets the same minimum, maximum, and size. But rather than aliasing the existing array, we make sure to initialize the _tallies pointer to a newly allocated array at line 15. In lines 16 and 17, we use a loop to copy the original array entries into the new array.

Destructors

Just as the constructors of a class are used for initializing the state of a newly created object, there is a method known as a ***destructor*** that is invoked each time an object reaches the end of its lifespan (e.g., a value variable that goes out of scope, or a dynamically-allocated object that is explicitly deleted). As is the case with the copy constructor, C++ will provide an implicit definition for a destructor if the programmer does not. The default behavior invokes the destructor upon each of the data members.

For many classes, such as our Point class, this implicit behavior suffices. However, ambiguity exists when pointers are used as data members. When an instance is being destroyed, should an object that it references be destroyed as well? With the default destructor, the memory for the pointer itself is reclaimed, but the object to which it points is not. For dynamically-allocated data such as the TallySheet's underlying array, we want to deallocate its memory as well. We can override the default behavior by providing an explicit destructor in the class definition. By convention, the destructor is a method whose name is the class's name preceded by a ~, taking no parameters, and providing no return value (not even **void**). Our TallySheet destructor appears as,

```
~TallySheet( ) {
  delete[ ] _tallies;          // deallocate the dynamically−allocated array
}
```

had we not explicitly deleted the dynamically-allocated array, it would remain stranded in memory indefinitely. This is in contrast to an array declared as a standard value variable, which is automatically reclaimed upon destruction of an instance.

The Assignment Operator

The assignment operator = is used to assign the left-hand operand the value of the right hand operator. For example, with integers variables we write x = y to set the value of variable x equal to the value of variable y. In Section 8.1 we emphasized that for the Point class, the command a = b assigns Point a the *value* currently held by b. That is a._x is assigned the value b._x, and a._y is assigned the value b._y. Technically, the = symbol is an operator and controlled by a method named **operator**=, just as the semantics of + is controlled by **operator**+. Yet C++ will automatically provide a default semantics if we do not implement **operator**=. The default is a member-by-member assignment, setting each data member of the left-hand operand to the value of the respective member of the right-hand operand. For our Point class, that is exactly the behavior we want, so we did not include an explicit implementation. If we had, it would appear as follows.

```
Point& operator=(const Point& other) {
  _x = other._x;
  _y = other._y;
  return *this;
}
```

The assignment of the members _x and _y is straightforward. The noteworthy aspect is the treatment of the return value. At first thought, it might seem that an expression a = b is an action, but should not have a resulting value. Yet it is supposed to have the new value as its result so that assignments can be chained as a = b = c. This treatment is similar to that of the chaining of stream operators, as originally demonstrated in Figure 11 on page 33. So we return the Point itself (as a reference, to avoid unnecessary copying). Internally, we use the keyword **this** to identify the Point instance being operated upon. C++'s treatment of **this** is akin to Python's **self** identifier. From a technical perspective, **this** is a pointer type in C++. Because we want to return the Point rather than a pointer to a Point, we dereference ∗**this** in the return statement.

While we did not have to explicitly provide the assignment operator for our Point class, greater care is necessary for classes such as TallySheet that make use of dynamically-allocated memory. The _tallies pointer on the left-hand side would become an alias for the array on the right-hand side, and the original left-hand array would be lost (i.e., a memory leak). Instead, we must ensure that the instances maintain their own arrays. Since, the existing array for the left-hand instance may not be the correct size for the renovated state, we deallocate the old array, and then reallocate a new array of the correct size. Then we copy the contents. The assignment operator for our TallySheet class appears as follows.

```
TallySheet& operator=(const TallySheet& other) {
  if (this != &other) {              // ignore self−assignments
    _minV = other._minV;
    _maxV = other._maxV;
    _size = other._size;
    delete[ ] _tallies;              // throw away old array
    _tallies = new int[_size];       // create new array with updated size
    for (int i=0; i < _size; i++)
      _tallies[i] = other._tallies[i];
  }
  return *this;
}
```

The core piece of code is very reminiscent of the commands used in the copy constructor and destructor, as we are in essence throwing away our old state and then resetting a new state. However, we wish to address one subtlety. Most of the body is shielded by the condition (**this** != &other). When a user invokes an expression such as s = t, **this** is the address of the left-hand operand while other is a reference to the right-hand operand (thus &other is the memory address of that instance). Typically, if a user is going to bother to do an assignment, the left-hand and right-hand operands will refer to different instances. But it is possible for someone to invoke what is known as a ***self-assignment*** such as s = s. While it is unlikely that a programmer would author such code, it is legal. More commonly, a self-assignment may occur when a programmer has two different references to the same object yet does not realize so. In some contexts, code written as s = t may actually involve a single instance.

If we recognize that we are being asked to assign an instance its own value, we may as well avoid doing unnecessary work. The conditional block serves this purpose (notice that the final return statement must still be executed, even for a self-assignment). Yet, our conditional treatment serves a far more important role than simply for improved efficiency. If we did not shield a self-assignment from this block of code, there are disastrous results. The problem is that we intentionally deleted the “old” array, then we create a new array and subsequently try to copy information from other._tallies into _tallies. While this if fine in the general case, if other happens to be the same instance, we will have just thrown away all our data and it is too late to recover it.

9 Generic Programming and Templates

The explicit type declarations of C++ introduces another challenge in comparison to Python. We often wish to write pieces of code that are *generic*, in the sense that the same commands can be applied to a variety of data types. For example, we might want a function that computes the minimum of two values, or which swaps two values, irregardless of the precise data types of those values. We may wish to have a class that manages data in a generic way, with flexibility as to the underling data type.

Code that is capable of working with a variety of data types is known as ***polymorphic***. In Python, polymorphism is supported directly through dynamic typing, as a programmer can write functions and classes without explicitly declaring the data types of the parameters. C++ supports polymorphism using a technique known as ***templating***. In this section, we discuss the use of templated functions and templated classes.

9.1 Templated Functions

In Python, it is easy to write a generic function that is capable of operating with a variety of data types. As an example, we might write a function for computing the minimum of two values as

```
def min(a, b):
  if a < b:
    return a
  else
    return b
```

If we send two integers as parameters, this function properly returns the smaller value. If we send two strings as parameters, this function returns the one that occurs first alphabetically. We do not need to explicitly declare the data types. So long as the < operator is well defined for the given data, the function works at run-time; if not, a run-time error will occur.

With the static typing of C++, we need additional support for writing such a general function. If we were only interested in such a function for integers, we might write

```
int min(const int& a, const int& b) {
  if (a < b)
    return a;
  else
    return b;
}
```

If we were only interested in such a function for strings, we might write

```
string min(const string& a, const string& b) {
  if (a < b)
    return a;
  else
    return b;
}
```

But we do not wish to write many different versions of such a function when one suffices. The challenge in C++ is that we must explicitly declare the type of all parameters and of the return type. Support for such generality in C++ is provided using a mechanism known as a template. We can define a function in a general way by declaring a placeholder for the type name immediately before the function definition. In the case of min, the templated function appears as follows.

```
template <typename T>
T min(const T& a, const T& b) {
  if (a < b)
    return a;
  else
    return b;
}
```

The first line designates the identifier T as a hypothetical type name. There is not actually a type with that name and we could have chosen any new identifier as such. This serves as what is known as the ***template parameter***. We use it in the signature on the second line to designate the return value and both parameters as having that same type.

When a call such as min(52,50) is attempted elsewhere in our program, the compiler tries to find a match for the template parameter T that will satisfy the compile-time checking. In this particular example, it will recognize that the call matches the function signature when using **int** in place of the template parameter T. If a call is made with two string instances as parameters, the compiler will resolve the template parameter as type **string**. After matching the template parameters, the compiler then attempts to compile the templated code. A compilation error would be reported at that time, for example if min were called on a data type that did not support the < operator used in the body of our function[6].

As a second example, we consider the basic task of swapping the values of two variables. In Python, this can conveniently be done with a simultaneous assignment statement, such as a,b = b,a. In C++, such a swap of values is typically accomplishes through use of a temporary variable using an approach like the following:

```
temp = a;
a = b;
b = temp;
```

However to write such code generically, we would need to make a formal declaration for variable temp and this depends on the data type being used. The <algorithm> library in C++ provides a generic implementation of a function with signature swap(a,b) that accomplishes this task. It can be implemented as a templated function as follows:

```
template <typename T>
void swap(T& a, T& b) {
  T temp(a);
  a = b;
  b = temp;
}
```

Notice that the template parameter is used not only in declaring the parameter types, but also in declaring the local variable temp. Also note that the parameters are passed by (non-const) references so that assignments within the function body affect the actual parameters of the caller.

9.2 Templated Classes

In our next example, we revise the TallySheet class from Section 8.6 to demonstrate how templating can be used in a *class* definition. Our original Python version was designed in a way so that it could track counts for either integers or one-character strings. It relied on run-time type-checking so that when a character is used as a value, it is converted to an appropriate integer when computing the proper array index. The C++ version of TallySheet sketched in the previous section assumed that the values to be tracked were integers. In particular, the constructor presumed that minVal and

[6]As a subtle aside, this particular implementation does not properly handle a syntax such as `min("hello","goodbye")` because the literals `"hello"` and `"goodbye"` are not technically strings. They are treated as character arrays, in this particular example, with different lengths. Furthermore, the < operator for arrays does not depend on the content of the arrays but on their memory addresses. More care would be required to add support for such a syntax.

maxVal were integers, as were the data members _minV and _maxV. In this section, our goal is to provide a templated version that can operate on integers or characters.

A complete version of the updated class is given in Figure 21. We wish to draw attention to a few aspects of the code. We begin by examining the data members declared in lines 4–7. Notice that _minV and _maxV are declared using the template parameter type rather than specifically an **int**. In contrast, _size is left as an **int** because the size of the array is an integer, even if the user is tracking characters from `'a'` to `'z'`. Similarly, we will maintain an array of integer counts, even if those counts represent the number of occurrences of a given character.

The standard constructor can be found at lines 10–14. Notice that the two parameters are typed as constant references to instances of the template type T in the signature. While the rest of this constructor is identical to the one given earlier in this section, there is an important subtlety. We initialize the integer _size to be the value (maxVal−minVal+1). When the parameters maxVal and minVal are integers, this is clearly a valid assignment. Less obviously, the assignment is valid even when maxVal and minVal are of type **char**. C++ allows us to subtract one character from another, producing an integer that is the distance between the two characters in the alphabet encoding.

The copy constructor, destructor, and assignment operators (lines 16–37) are almost verbatim from the previous section, except for the use of the template parameter when designating other variables of type TallySheet<T> as opposed to the simpler TallySheet; we will address this issue further in Section 9.3. We see the use of the template type T for the parameters of the public methods increment and getCount, as those are values in the user's domain. As we did in Python, we convert such a value to the integer index into the underlying array using a private _toIndex method. At line 76, that method relies on the use of subtraction for type T, similar to our computation of the size in the constructor.

The _makeLabel method uses a stringstream to convert the index into a string that appropriately matches the user's domain value. That technique was introduced in Section 6.6. At line 82, we use the syntax T(...) to ensure that the label is formatted as the appropriate data type (e.g., **int**, **char**). Because ind is always an **int**, the intermediate expression ind + _minV will also be an **int**, even when _minV is a **char**. By surrounding that value with the syntax T(...), it will be correctly converted to the desired type before being inserted into the stream.

9.3 Using Templated Functions and Classes

When calling a templated function, the standard calling syntax will often suffice. For example, we may use the expression min(52, 50) with our templated min function, and the compiler will automatically determined that **int** should be used for the template parameter. However, the syntax min(1, 0.5) would not be legal. The first operand is an **int** and the second is a **double**. While it is permissible to send an **int** to a function that expects a **double** or to send a **double** to a function that expects an **int**, C++ does not know whether we intended for the template parameter to be **int** or **double**. In this case, we can make this explicit in the calling syntax as min<**double**>(1, 0.5). Similarly, in the footnote on page 49, we discussed why a syntax min(`"hello"`,`"goodbye"`) was not supported by our templated function because the literals are not technically strings. However, we could legally use the syntax min<**string**>(`"hello"`,`"goodbye"`), as the explicit template parameter make the calling signature clear, and string parameters are implicitly created based on the literals.

For a templated class, such as our TallySheet, the declaration of the template parameter is necessary. That is, it is illegal to make a declaration

```
TallySheet frequency('A','Z');     // illegal: must specify template parameter
```

```
template <typename T>
class TallySheet {
  private:
    T _minV;
    T _maxV;
    int _size;
    int* _tallies;

  public:
    TallySheet(const T& minVal, const T& maxVal) :
            _minV(minVal), _maxV(maxVal), _size(maxVal−minVal+1), _tallies(new int[_size]) {
      for (int i=0; i < _size; i++)
        _tallies[i] = 0;
    }

    TallySheet(const TallySheet<T>& other) :
          _minV(other._minV), _maxV(other._maxV), _size(other._size), _tallies(new int[_size]) {
      for (int i=0; i < _size; i++)
        _tallies[i] = other._tallies[i];
    }

    ~TallySheet( ) {
      delete[ ] _tallies;
    }

    TallySheet<T>& operator=(const TallySheet<T>& other) {
      if (this != &other) {              // ignore self−assignments
        _minV = other._minV;
        _maxV = other._maxV;
        _size = other._size;
        delete[ ] _tallies;              // throw away old array
        _tallies = new int[_size];       // create new array with updated size
        for (int i=0; i < _size; i++)
          _tallies[i] = other._tallies[i];
      }
      return *this;
    }

    void increment(const T& val) {
      int ind = _toIndex(val);
      if (!(0 <= ind && ind < _size))
        throw range_error("Parameter out of range");
      _tallies[ind] += 1;
    }
```

Figure 21: Templated TallySheet class (continued on next page).

```
    int getCount(const T& val) const {
      int ind = _toIndex(val);
      if (!(0 <= ind && ind < _size))
        throw range_error("Parameter out of range");
      return _tallies[ind];
    }

    int getTotalCount( ) const {
      int sum = 0;
      for (int i=0; i < _size; i++)
        sum += _tallies[i];
      return sum;
    }

    void writeTable(ostream& out) const {
      out << "Value  Count Percent \n----- ------ -------\n";
      int total = getTotalCount( );
      if (total == 0)
        total = 1;                    // avoid division by zero

      for (int ind=0; ind < _size; ind++) {
        string label = _makeLabel(ind);
        int count = _tallies[ind];
        float pct = 100.0 * count / total;
        out << setw(5) << label << setw(7) << count;
        out << fixed << setw(7) << setprecision(2) << pct << endl;
      }
    }

  private:
    int _toIndex(const T& val) const {
      int i = val - _minV;
      return i;
    }

    string _makeLabel(int ind) const {
      stringstream converter;
      converter << T(ind + _minV);
      string output;
      converter >> output;
      return output;
    }
};
```

Figure 21 (continuation): Templated TallySheet class.

That is, even though the apparent min and max value sent to the constructor are characters, the compiler requires that the template parameter be clearly defined as part of the data type. The correct declaration for such an instance is:

```
TallySheet<char> frequency('A','Z');
```

The official type of our frequency variable is TallySheet<**char**>. Formally, we get a distinct type definition for each instantiation of the template parameter (ie TallySheet<**char**> is distinct from TallySheet<**int**>). We see this issue arise in our original code for the class, as given in Figure 21. For example, the copy constructor definition at line 16 takes a parameter that must be a TallySheet<T> instance, matching the same template parameter as the given instance. This makes sense, as we cannot make a TallySheet<**char**> instance as a copy of a TallySheet<**int**> instance. We see similar usage in the signature for the assignment operator at line 26, as both the parameter and the return type are explicitly TallySheet<T>.

10 C++ Containers and the Standard Template Library

In Python, most container types are able to handle heterogeneous data. For example, it is possible to have a list composed as [5, 'alpha', {'a': 3, 'b': 5}], having three elements, the first being an integer, the second a string, and the third a dictionary. This is possible in Python because the **list** instance is stored as a sequence of *references* to objects (each reference has equivalent size).

In C++, container types require a static declaration of the element type being stored, and thus they are homogeneous by nature. However, most container definitions can be applied to a variety of element types and are implemented using templates for generality (templates were discussed in Section 9). The most commonly used data structures have been implemented as part of the ***Standard Template Library***, or ***STL*** for short. In this section, we give a brief introduction to the main features of that library. In Section 10.1, we introduce the **vector** class as an example of a typical STL class. We give an overview of several other classes in Section 10.2. Finally, in Section 10.3, we discuss the concept of an ***iterator***, which is used throughout the STL framework.

10.1 The vector Class

We begin by examining the **vector** class, which is defined in the <**vector**> library. A vector is used to maintain an ordered sequence of values. Internally, the values are stored sequentially in an array, and the class ensures that the underlying array is resized as necessary when elements are added to the vector. A C++ **vector** is probably the closest analog to Python's **list** class, but we more accurately compare it to the more specialized **array** class in Python because a vector is homogeneous and stored as a collection of *values* (rather than as a collection of references to values, as with Python's **list**). To demonstrate its basic usage, we consider the following sample code.

```
vector<string> groceries;
groceries.push_back("bread");
groceries.push_back("milk");
groceries.push_back("cheese");
cout << groceries.size( ) << endl;          // will be 3
cout << groceries[2] << endl;               // will be "cheese"
groceries[1] = "juice";                     // replaces "milk"
```

We start by noting that **vector** is a templated class. In this case, we are declaring a vector of strings. By default, the newly instantiated vector is empty. The push_back method is the C++ analog of Python's append, adding the new value to the end of the sequence. The length of the vector is returned by the size( ) method. As is the case with Python, elements are zero-indexed and can be accessed with a syntax such as groceries[2]. However, unlike Python, C++ does not check the validity of an index at run-time; it simply trusts the programmer (with potential disaster if the programmer is wrong). A safer (yet slower) way to access an element in C++ is with the syntax groceries.at(2); this version performs an explicit run-time check of the given index, throwing an out_of_range exception when warranted. There are several other behaviors supported by the **vector** class; we refer the interested reader to more detailed documentation elsewhere.

10.2 Other STL Classes

The Standard Template Library contains definitions for many other useful container classes. We focus primarily on those that closely mirror Python's built-in container classes, as shown in Figure 22. A great advantage of the STL framework is that all of its classes share some common behaviors and interfaces. For example, the syntax data.size( ) is supported uniformly by all of these data types to report the number of items stored in the container. In Section 10.3, we will discuss how all containers provide a standard syntax for iterating over their elements.

The **string** class is formally part of the STL, although it is designed specifically for a sequence of characters (as opposed to a sequence of arbitrary type). In contrast to Python's immutable **str** class, a C++ string is mutable and supports methods such as append, insert and erase that modify the contents of a string. The **vector** class, as introduced in Section 10.1, is an array-based sequence of elements with support for arbitrary capacity. While the C++ **vector** class is quite similar to the Python **list** class, it is important to note that there exists a **list** class in C++ supporting an entirely different concept. The C++ **list** class represents an ordered sequence of elements, but one that is stored internally as what is known as a "linked list," rather than the array-based **vector**.

Both Python and C++ support a **set** class, but they use different underlying data structures and provide different guarantees in terms of ordering and efficiency. Python's sets are implemented using an approach known as ***hashing***, described in Chapter 13.3.2 of our book. This approach provides constant-time operations in general, but the elements of the set are not well-ordered. In contrast, the C++ **set** class represents an *ordered* set, implemented using a balanced binary search tree similar to that described in Chapter 13.4 of our book. For this reason, the element-type for a set must define a total ordering, by default based on an implementation of **operator**<.

We relied on the **set** class in an earlier piece of code in this document, namely when implementing the DeluxeTV class in Figure 12 on page 34. At line 3 of that code, we declare instance variable _favorites having templated type **set**<**int**>. We add an element to the set at line 11 using the insert

C++ Type	Description	Python analog
string	character sequence	**str**
vector	array-based expandable sequence (homogeneous)	**list** or **array**
list	linked sequence (homogeneous)	
set	ordered set of unique elements (homogeneous)	**set**
map	associative mapping (homogeneous)	**dict**

Figure 22: Commonly used classes from C++'s Standard Template Library (STL).

method, and we remove an element at line 13 using the erase method. We will discuss the rest of that example after introducing the concept of iterators in Section 10.3.

Finally, the C++ **map** class is the analog to the Python **dict** class. Rather than storing a set of elements, it manages a collection of key-value pairs. As such, it requires two template parameters, the first specifying the key-type and the second the associated value-type. As an example, we could have implemented the TallySheet to keep track of frequencies by maintaining a map from some key type to an integer frequency. For example, if counting characters, we may declare **map**<**char**, **int**> _tallies. As is the case with sets, C++ uses balanced binary trees to implement maps, and the key type must define a total ordering, typically with **operator**<.

10.3 Iterators

Because a string or vector instance is stored in an underlying array, integer indices can be used to efficiently describe the position of an element to be accessed. For most other STL container types, integer indices are not supported. This is typically because a container is not inherently ordered or because the underlying storage mechanism is not consistent with such a convention (for example, a **list** instance implemented as a linked list, or a **set** instance implemented as a balanced search tree).

For this reason, all STL containers (including strings and vectors) provide support for describing a "position" of an element using an instance of an **iterator** class. Iterator instances cannot be directly created by a user, rather they are initially returned by some method of the class. These iterators are then used to identify a particular element or location within a container in the context of parameters and return values. Conceptually, they are similar in purpose to a pointer, but they are not necessarily a direct representation of a memory address. To explore the syntactic use of iterators, we revisit a code fragment from our DeluxeTV class definition in Figure 12.

```
17      set<int>::iterator result = _favorites.upper_bound(_channel);
18      if (result == _favorites.end( ))
19        result = _favorites.begin( );      // wrap around to smallest channel
20      setChannel(*result);
```

The result variable is defined to have type **set**<**int**>::iterator. Formally, this is an instance of an iterator class that is nested within the scope of the **set**<**int**> class. This is different from other iterator classes, such as **set**<**string**>::iterator or **vector**<**int**>::iterator.

The value of result is initialized to the result of the call to upper_bound, a method specific to the **set** class. Before explaining the semantics of upper_bound, we describe a related method named find supported by sets. A call such as _favorites.find(value) checks to see if the given value is contained in the set, akin to the __contains__ method in Python. If found, an iterator representing that element's position in the set is returned; if not, the sentinel end( ) is returned, by convention, to designate the lack of such an element. In contrast, the upper_bound method is one that relies on the fact that C++'s sets are *ordered*. The formal semantics of a call to upper_bound(value) is to locate the smallest value in the set, if any, that is strictly greater than the parameter. So in the context of DeluxeTV, we are looking for the next favorite that is found when moving upward from the current channel setting. If there is no element of the set with a value greater than _channel, the end sentinel is returned, just as is done with find. We detect that case at line 18. For the television model, our goal was to then wrap around to the smallest of all favorite channels. We accomplish this goal at line 19 by calling another method named begin. In some sense, this is the opposite of end, except that the result of begin is an iterator to the first actual position in the set (rather than a sentinel). In the case of sets, the ordering begins with the least element.

The final lesson portrayed by the above code involves the use of syntax *result at line 20. In context, the call to setChannel is used to implement the change of channels and the parameter to that call must be the desired channel value. Just as there is a distinction between a pointer and the value to which it points, we must make a distinction between an iterator and the underlying element to which it refers. In this context, result is the iterator while *result is the corresponding **int** value from the set, namely the favorite channel number.

Iterators can also be used to iterate through all elements of an STL container (hence, the term *iterator*). As an example, here is code that prints out all favorite channels for our DeluxeTV using the ++ operator to advance from one position to the next.

```
for (set<int>::iterator walk=_favorites.begin( ); walk != _favorites.end( ); ++walk)
  cout << *walk << endl;
```

Note the significance between the asymmetric convention of begin and end. So long as the iterator is not equal to the end sentinel, it represents the position of an actual element that can be printed. When walk becomes equivalent to the end, the loop terminates; note that for an empty set, the result of begin( ) is precisely that of end( ) and so there are zero iterations of the loop. This form of loop is supported by all STL containers. We note the correspondence between going from begin up to but not including end, just as Python handles ranges start:stop. Iterators for many containers also support backwards iteration with the −− operator. Indexed classes such as **vector** and **string** support ***random-access*** iterators that allow arbitrary step sizes using arithmetic such as begin( )+5 or end( )−3.

Finally, we note that an assignment such as *walk = val can be used to modify an element of a container in-place. However, if a container is designated as read-only with **const**, modification of elements is not allowed. For this circumstance, all STL containers support a second class named const_iterator, that is similar to iterator but without the ability to modify the contents of the underlying container. Note well that a const_iterator is not the same as a **const** iterator, as the former can be incremented and decremented but without modifying the underlying container, while the latter can modify the underlying container but cannot be incremented or decremented.

11 Error Checking and Exceptions

11.1 Overview

As is the case with Python, C++ provides support for throwing and catching exceptions as a way to handle exceptional cases that arise at run-time. C++ defines an exception type that is the base class for a hierarchy of standard exception types defined in the <stdexcept> library. For example, there is an out_of_range exception in C++ that is similar in purpose to Python's IndexError, and a domain_error in C++ that is similar to Python's ValueError. Programmers are also free to define their own subclasses derived from exception. In Sections 11.2 and 11.3, we will give a brief overview of the syntax for throwing and catching exceptions in C++. However, exceptions are not the only way that C++ manages errors. In Section 11.4 we will discuss several other approaches for error-handling.

11.2 Throwing an Exception

One of the examples we used in Python was that of a sqrt function that throws an exception when the argument is a negative number. The implementation for such a function might begin as follows:

```
def sqrt(number):
  if number < 0:
    raise ValueError('number is negative')
```

Python uses the keyword **raise** to throw an exception, with the remaining argument being an instance of the exception class to be thrown. In this example, ValueError('number is negative') denotes the construction of a new instance of type ValueError, with the constructor for that class accepting an error message as a string parameter. The syntax in C++ is quite similar, except that we use the keyword **throw** rather than Python's **raise**. The same code fragment would appear as follows in C++:

```
double sqrt(double number) {
  if (number < 0)
    throw domain_error("number is negative");
```

11.3 Catching an Exception

Exceptions can be caught and handled in both Python and C++ using a **try** construct. In Python we saw that the corresponding handlers were labeled with the keyword **except**. In C++, the corresponding handlers are labeled with the keyword **catch**. As with Python, there can be any number of **catch** clauses as part of a **try** construct, each looking for one particular type of error, and there can be a final clause that catches anything else that was not yet caught. A generic example of the C++ syntax is as follows

```
try {
  // any sequence of commands, possibly nested
} catch (domain_error& e) {
  // what should be done in case of this error
} catch (out_of_range& e) {
  // what should be done in case of this error
} catch (exception& e) {
  // catch other types of errors derived from exception class
} catch (...) {
  // catch any other objects that are thrown
}
```

If an exception is encountered, it will be handled by the first clause with a matching type declaration. The final clause with parameter ... will match any object. If there were not such a final clause, and an exception occurs that does not match any of the existing clauses, that exception would be propagated to any outer nested scope, where it might be caught or if uncaught will cause the program to terminate. Note that for all but the final case, the exception handler clause has access to the declared variable e, with local scope, that is a reference to the exception instance that has been caught. Additional information about that exception can be determined, for example with the call e.what( ), which returns the string message used when the exception was instantiated.

11.4 Other Approaches to Error-Handling

Despite having support for formally throwing and catching exceptions, this is not the only mechanism used by C++ for managing exceptional cases. In fact, due to legacy code, other techniques are far more commonly seen in the standard packages.

As an example, consider what happens when you attempt to divide an integer by zero. In Python, this causes a formal ZeroDivisionError to be raised, and a programmer could write code to catch such an error. In C++, the behavior of such a division by zero is inherited from treatment in C, and so it causes a run-time error but not formally by means of a **throw** statement. As a result, it cannot in general be caught within a **try** construct; a programmer who is concerned about the validity of a division must check the denominator before performing the operation to avoid any such error.

Another legacy behavior from C is the treatment of array indexing. In Section 8.5, we discuss how C++ computes the address of an array entry by using the index to compute a relative offset from the beginning of the array. We also noted that C++ does not explicitly check whether an index is within range for a declared array, and so access to an expression like measurements[k] for a k that is too larger or is negative will likely have dire, but unpredictable consequences. In Python, indices are always validated at run-time. A C++ programmer could also do such checks, to ensure that an index always lies in the appropriate range. However, performing such a check at run-time is a burden on the system. If a programmer is confident in the logical design of a program, there is no need to explicitly perform such checks. But logical errors involving invalid indices are difficult to detect and diagnose. Since both strings and vectors in C++ are implemented using C-style arrays internally, the same issues arise as to how to handle error-checking of indices. The designers of those classes struck a compromise. In Section 10.1 we differentiated between the two syntaxes measurements[k] and measurements.at(k) that are supported by strings and vectors. The first is the more efficient version that does not explicitly error-check the index value (but which has indeterminate behavior when errant). In contrast, the at method performs a run-time check of the given index, formally throwing a out_of_range exception when errant.

Yet another mechanism for detecting errors is used by C++ for most I/O tasks such as reading formatted data or working with files. As a basic example, we revisit the goal of reading a number from 1 to 10 from a user. In Python, this is accomplished with the command number = **int**(**raw_input**(`'Enter a number from 1 to 10: '`)), noting that the call to **raw_input** might throw exceptions if there were failures to read data from the user, and that the call to the **int** constructor will fail if the string returned by **raw_input** is not a legitimate representation of an integral value. In Python, more robust code is based on catching the various exceptions that occur with appropriate remedy. Our more robust C++ version of that task was given in Figure 8 on page 27. By default, C++ uses a mechanism based on what are known as ***error flags***. A command such as **cin** >> number does not formally throw any exceptions. Instead, if various problems arise, they are tracked through extra fields associated with the stream instance. We can check for error cases after the command by polling those various flags using calls such as **cin**.fail( ), **cin**.eof( ), **cin**.bad( ), each of which denotes one of a certain set of possible errors that have occurred. Once an error flag has been set, it is up to the program to find remedy, and to explicitly unset that flag if the stream is going to continue to be used. Please review the prose on page 27 for further explanation of that example.

Error flags are similarly used in C++ when working with file streams. As a tangible example, Figure 8.6 on page 283 of our book gives a Python function for opening an existing file based on a filename given by the user, with appropriate error-checking to ensure success. In translating that example to C++, we cannot return a local variable for the resulting file stream, as file streams

```
void openFileReadRobust(ifstream& source) {
  source.close( );                               // disregard any previous usage of the stream
  while (!source.is_open( )) {
    string filename;
    cout << "What is the filename? ";
    getline(cin, filename);
    source.open(filename.c_str( ));
    if (!source.is_open( ))
      cout << "Sorry. Unable to open file " << filename << endl;
  }
}
```

Figure 23: A C++ function for robustly opening a file with read-access.

cannot be passed by value, so we use a signature in which the caller passes an uninitialized file stream as a parameter, and the function's responsibility is to open the underlying file. Our C++ variant is given in Figure 23. Notice that we rely on polling the result of is_open( ). This method is supported by the file stream classes, in addition to the other accessors such as eof and bad, inherited from the more general stream classes.

12 Managing Large Projects

Thus far, we have assumed that source code for a project is contained in a single file. For medium- and large-scale software projects, it is more common to have source code divided among many different files. There are several advantages to such a design. A multi-file structure can support a more natural decomposition for complex project, while also allowing a team of developers to edit different files concurrently. Furthermore, having components of a program in different files provides a better structure for version control and reuse of source code across projects.

As a simple demonstration of a more typical C++ structure, we revisit our first example from Figure 1 on page 7, in which we define a gcd function as well as a main function for testing it. A multi-file version of that project might appear as in Figure 24. In our new design, we have intentionally separated the implementation of the gcd function from the piece of code (in this case, the main function) that relies upon the gcd function. To coordinate the interactions between those two, we created the file `gcd.h` which is known as a ***header file***; note that we were not required to name the file `gcd.h`, but this choice is reasonably conventional. The purpose of that file is not to implement the gcd function, but to provide a formal definition of its interface. This is accomplished at line 3 of that file, which gives the full signature of the function but without a subsequent function body (we will discuss the purpose of the other lines in `gcd.h` later).

With the information encapsulated in the header file, we are able to independently define the two other components of our software. We use the **#include** `"gcd.h"` directive from within the other files to load this definition. This is the same directive we have already seen for loading definitions from standard libraries, such as with <iostream> at line 2 of `gcdTest.cpp`. The distinction between the quotation marks around `"gcd.h"` versus the angle brackets around <iostream> is because we presume that the `gcd.h` file is stored in the same directory as the file `gcdTest.cpp`, whereas the iostream library definitions are installed elsewhere on the system (in a path known to the compiler). With `"gcd.h"` included from within `gcdTest.cpp`, the compiler is able to verify the proper usage `gcd.h`

```
1  #ifndef GCD_H
2  #define GCD_H
3  int gcd(int u, int v); // forward declaration
4  #endif
```

`gcd.cpp`

```
 1  #include "gcd.h"
 2
 3  int gcd(int u, int v) {
 4    /* We will use Euclid's algorithm
 5        for computing the GCD */
 6    int r;
 7    while (v != 0) {
 8      r = u % v;    // compute remainder
 9      u = v;
10      v = r;
11    }
12    return u;
13  }
```

`gcdTest.cpp`

```
 1  #include "gcd.h"
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main( ) {
 6    int a, b;
 7    cout << "First value: ";
 8    cin >> a;
 9    cout << "Second value: ";
10    cin >> b;
11    cout << "gcd: " << gcd(a,b) << endl;
12    return 0;
13  }
```

Figure 24: GCD project with source code consisting of three files.

of the call to the `gcd` function at line 11. Note well that it does not matter in that context how the `gcd` function is implemented, only that the type of parameters and return value are known.

The purpose of the `gcd.cpp` file is to implement the `gcd` function. We include the header file at line 1 to ensure consistency of the signature, although this is not technically required.

12.1 Compilation and Linking

In the original version of our project, with all code within a single-file named `gcd.cpp`, we compiled the program into an executable named `gcd` with the following command.

```
g++ -o gcd gcd.cpp
```

With our new design, the appropriate compiler command for building an executable named `gcd` is the following.

```
g++ -o gcd gcd.cpp gcdTest.cpp
```

Notice that we provide both files `gcd.cpp` and `gcdTest.cpp` as inputs to the compiler. This is because each of them includes part of our implementation. In contrast, we do not explicitly indicate the header file `gcd.h`; the definitions from within that file are already explicitly included by the **#include** directive when compiling the other files.

To build the executable, both parts of the source code must be individually compiled, and then those parts are combined to create the final executable. Formally, these are separate stages of the compilation process. Since the implementation of the function in `gcd.cpp` does not overtly depend upon the code that calls the function, the compiler can analyze it independently, checking for validity and converting the high-level commands into appropriate machine code. Similarly, the commands of the `main` function in `gcdTest.cpp` can be independently compiled into machine code without regard to the details of the `gcd` function body.

The final stage of the compilation process is known as ***linking***. Once the individual pieces have been converted to machine code, the system must assemble them into a single, coherent executable that can be executed on the system. The primary goal of the linking stage is to ensure that all necessary functions have been defined in precisely one of the components. That is, if we tried to compile the file `gcdTest.cpp` without `gcd.cpp`, the linker would report an error that it cannot find the implementation of the function `gcd` that is being called from within `main`. In contrast, if we were to attempt to compile `gcd.cpp` without `gcdTest.cpp`, the linker would complain that it cannot find a `main` function, a requirement for any executable.

The given `g++` command from above performs both the compilation and linking phases by default. That said, it is possible to request that a component be compiled, but to defer the linkage. For example, to compile the `gcd` function, but not yet link it to any executable, we could execute the command

```
g++ -c gcd.cpp
```

The `-c` flag in the command is what designated our desire to perform compilation only. The result of a successful compilation in this case is a new binary file named `gcd.o` that is known as ***object code***. One advantage of having this object code stored in a file is that it allows us greater reuse without additional compilation. For example, if we wanted to use our `gcd` function in several projects, each of those could make use of our pre-compiled object code rather than re-compiling from the original source code. In our example, if we presume that we have preliminary created both `gcd.o` and `gcdTest.o` as object code, we can invoke the final linking to produce an executable with the command.

```
g++ -o gcd gcd.o gcdTest.o
```

For large-scale problems, there is another great advantage of separating out the compilation of object code from that of the linking phase. If you envision a project that might be composed of hundreds if not thousands of files, the full compilation process is a timely one. During the development cycle, it is common to have compiled the program, to make changes to one or more of the source code files, and then to re-compile the entire project. However, with good modular design, a change to one piece of source code should not effect most of the other components. In this case, we will need to re-generate the object code for the modified source code, and perhaps a few other dependent pieces, but the majority of the components will not need to be rebuilt from scratch. Instead, those few components can be recompiled into object code, and then all of the object code can be relinked to form a new executable. Although the linking phase requires some work, it is not nearly as time-consuming as the original compilation.

Finally, we note that for larger problems, there are other tools to assist developers in managing their projects. For example, many Integrated Development Environments (IDEs) will keep track of which pieces of source code have been modified, and which need to be re-compiled when building a new executable. One of the classic tools for developers in managing the (re)building of a project is a program known as `make`. This program relies upon a configuration file for the project, conventionally named `makefile`. The makefile designates what components comprise the project, and upon which pieces of source code each component depends. The `make` command causes a re-build of the entire project, but relying on the file-system timestamps to determine which pieces of source code have been edited since the previous build.

12.2 Avoiding Multiple Definitions

Header files are typically used to provide formal declarations of functions or classes. It seems pretty clear that you would not want to include two conflicting definitions for something. As it happens, it is also illegal to repeat the *same* definition for something more than once. While this might seems easy to avoid, there are several pitfalls when including header files throughout a larger project. For example, assume we have an application that will rely on our gcd function, as well as a Fraction class. Source code for that project might naturally start with the following declarations.

```
#include "gcd.h"
#include "Fraction.h"
```

The problem is that the definition of the Fraction class might also depend upon use of the gcd function for reducing fractions to lowest terms. So it is possible that there might be an **#include** "gcd.h" within the Fraction header file. If this were the case, then the contents of gcd.h are inherently being included twice in the above code.

To properly avoid errant repetitions of definitions from a header file, those files take advantage of other preprocessor directives, as shown in our gcd.h file from Figure 24. These directives, beginning with #, are not formally C++ syntax, rather a separate set of commands that are understood by the compiler (as with **#include**). Our gcd.h file begins with the **#ifndef** GCD_H directive on line 1 paired with the **#endif** directive on line 4. The **#ifndef** syntax is short for "if not defined." It looks for whether the term GCD_H has been previously defined, only using the body of that conditional when it has not been defined. In this context, the first time we included the contents of gcd.h, no such symbol GCD_H has been defined, and so we will include the body of that conditional (lines 2 and 3). The purpose of the **#define** GCD_H directive on line 2 is to introduce the symbol GCD_H to the compiler so that the **#ifndef** directive at line 1 will fail for any subsequent inclusions of gcd.h. We will use a similar guard mechanism for all header files in a larger project, choosing a distinct symbol for each (e.g., GCD_H).

Line 3 of the file gcd.h is a forward declaration of the gcd function. This defines the signature of the function, yet we do not explicitly include the body of the function in that file. The forward declaration has the relevant information for doing compile-time checking, for example when making the call to gcd from within gcdTest.cpp, so the function body is not needed. More importantly, it is important that we do not place the body of the gcd function inside the header file. Otherwise, there would be a problem if two or more .cpp files include such a header. Each .cpp file is independently compiled into object code, and so the **#ifndef** guard would allow the definition to be read once for each .cpp file. Having the forward declaration in each is permissible, but if the function body were included, the compiled version of that body would be embedded within the object code associated with each .cpp file. This in turn would cause a linking-error when the object codes were combined, as there would be duplicate definitions for the function to contend with. In our actual project, it is only the compiled version of gcd.cpp that has the implementation of the gcd function, so there is no ambiguity when compiling and linking.

12.3 Namespaces

Another concern with larger projects is that several components may wish to use the same identifier for a feature such as a variable, function, or class. If all components were authored by the same team, perhaps such naming conflicts could be avoided. However, often we need to rely upon software packages written by others that cannot be modified. For example, a civil engineering application might need to include an architecture package that defines a Window class (i.e., a plane of glass) and a graphical user interface package that defines its own Window class (i.e., a rectangle on a computer screen). Including such conflicting definitions for the identifier Window would be illegal.

Python tackles the issue of naming-conflicts by supporting two forms for import. A command such as **import** architecture introduces the name architecture into the default namespace, as an identifier for the imported module. Then, a qualified name architecture.Window can be used to describe the Window class from that module. In contrast, a syntax such as **from** architecture **import** Window introduces the name Window directly into the default namespace. However, this form would be conflicting with another command **from** gui **import** Window. In C++, name conflicts can be mitigated by organizing related definitions into a separate namespace using a syntax such as the following:

```
namespace architecture {
  class Window {
     ...
  };
  class Building {
     ...
  };
}
```

After this definition, the Window class can be identified with a qualified name architecture::Window. This style would allow another part of a program to include the architecture code and to include the gui code, while being able to differentiate unambiguously between architecture::Window and gui::Window. At the same time, it is possible for a programmer to introduce definitions from such a namespace into the default namespace. For example, following the above definition, a subsequent command **using** architecture::Window; would introduce the architecture::Window definition into the default namespace with the unqualified name Window (akin to **from** architecture **import** Window in Python.). All definitions from a namespace can be added to the default namespace, for example using the command **using namespace** architecture; (akin to from architecture import * in Python).

We have already seen such a use of namespaces in our C++ code. By default, most standard libraries in C++ introduce their definitions into a special namespace identified as std, rather than directly in the default namespace. For example, the <iostream> library defines the streams **cin** and **cout** into the std namespace. Since the fully-qualified names std::**cin** and std::**cout** require extra typing, programmers often bring everything from the standard namespace into the default with the command **using namespace** std; such as at line 3 of our `gcdTest.cpp` program in Figure 24.

12.4 Managing Class Definitions

In Section 12.2 we emphasized that header files should provide function definitions, but not implementations for those functions. The rule is somewhat different in regard to member functions of a class definition. The implementation for those functions can either be given directly within the context of the class definition or externally. As a demonstration, we revisit the robust definition of the Point class as originally shown in Figures 10 and 11 in Section 7.2. Our revised version has been divided into two files `Point.h` and `Point.cpp`, as shown in Figures 25 and 26 respectively.

The header file defines the Point class, specifying the data members and the signatures of the member functions. Notice that for some simple functions, such as getX and setX, we have embedded the function bodies directly within the header file. Formally, those are processed by the compiler so that they become in-lined code at run-time, rather than a formal function call. In most other cases, we defer the implementation of the function to the `Point.cpp` file. Note well that we chose note to

```
 1 #ifndef POINT_H
 2 #define POINT_H
 3 #include <iostream>       // need ostream definition for operator<< signature
 4
 5 class Point {
 6 private:
 7   double _x;
 8   double _y;
 9
10 public:
11   Point(double initialX=0.0, double initialY=0.0);
12   double getX( ) const { return _x; }                    // in-lined function body
13   void setX(double val) { _x = val; }                    // in-lined function body
14   double getY( ) const { return _y; }                    // in-lined function body
15   void setY(double val) { _y = val; }                    // in-lined function body
16   void scale(double factor);
17   double distance(Point other) const;
18   void normalize( );
19   Point operator+(Point other) const;
20   Point operator*(double factor) const;
21   double operator*(Point other) const;
22 };    // end of Point class
23
24 // Free-standing operator definitions, outside the formal Point class definition
25 Point operator*(double factor, Point p);
26 std::ostream& operator<<(std::ostream& out, Point p);
27 #endif
```

Figure 25: `Point.h` header file for our Point class.

state **using namespace** std from within the header file, to avoid polluting the default namespace for others including this file. As a result, we explicitly designated the type std::ostream at line 26.

Looking at Figure 26, we see the remaining implementations of the functions from the Point class. Since this file is not directly being included by others, we promote the std namespace to the default at line 4 for our convenience. We wish to draw attention to the form of the signatures in this file, for example, that of Point::scale at line 8. Because this code is not formally included within the scope of the original Point class definition, we cannot simply write the signature as

```
void scale(double factor)
```

That would appear to be the signature of a stand-alone function named scale, rather than the member function of the Point class with that name. To properly declare that we are implementing the member function, we must use the qualified function name Point::scale to re-establish the proper scope. Having done so, the body itself is interpreted with the proper context. Therefore, we are able to access data members such as _x and _y, as done on lines 9 and 10. Notice that at line 20, within the body of the normalize function, we are able to call the distance method within need to qualify it as Point::distance because the proper scope has already been established for the body.

```cpp
#include "Point.h"
#include <iostream>              // for use of ostream
#include <cmath>                 // for sqrt definition
using namespace std;             // allows us to avoid qualified std::ostream syntax

Point::Point(double initialX, double initialY) : _x(initialX), _y(initialY) { }

void Point::scale(double factor) {
  _x *= factor;
  _y *= factor;
}

double Point::distance(Point other) const {
  double dx = _x - other._x;
  double dy = _y - other._y;
  return sqrt(dx * dx + dy * dy);         // sqrt imported from cmath library
}

void Point::normalize( ) {
  double mag = distance( Point( ) );      // measure distance to the origin
  if (mag > 0)
    scale(1/mag);
}

Point Point::operator+(Point other) const {
  return Point(_x + other._x, _y + other._y);
}

Point Point::operator*(double factor) const {
  return Point(_x * factor, _y * factor);
}

double Point::operator*(Point other) const {
  return _x * other._x + _y * other._y;
}

// Free-standing operator definitions, outside the formal Point class scope

Point operator*(double factor, Point p) {
  return p * factor;                             // invoke existing form with Point as left operand
}

ostream& operator<<(ostream& out, Point p) {
  out << "<" << p.getX( ) << "," << p.getY( ) << ">";       // display using form <x,y>
  return out;
}
```

Figure 26: `Point.cpp` source code for our Point class.

As a final comment, we draw attention to our treatment of the two stand-alone functions, namely **operator*** and **operator**<<, that provide support for operator usage when the Point is a right-hand operand. In our original treatment of the Point class, we explained the need for having those definitions outside the formal class. They remain as such in our new design, with forward declarations given at lines 25 and 26 of `Point.h` and implementations given at lines 39–46 of `Point.cpp`. Because these are not formal member functions, note that we do not designate Point:: scope in their signatures. For example, line 43 of `Point.cpp` names the function **operator**<< rather than Point::**operator**<<. In similar spirit, line 39 defines **operator***, not to be confused from the forms of Point::**operator*** that are given at lines 29 and 33.

12.5 Managing Templated Classes

The conventions are different for *templated* classes in a multi-file project. The main issue is that templated code is not pre-compiled into independent object-code, because the underlying machine code depends intricately on the actual data type supplied as the template parameter. Therefore, templated code is compiled as needed when instantiated from other contexts. As a result, we use a slightly different convention for embedding the source code within separate files.

There remains a choice of whether to embed implementations for member functions withing the formal class definition or externally. However, even when external to the class definition, the implementations must formally be included (directly or indirectly) as part of the *header* file, rather than as a separately compiled `.cpp` file. Still, to ease in the separation of the interface and implementation, it is a common convention to have a separate `.h` file in the style of the standard class, and the bodies in another file using a special suffix such as `.tcc`. That is to differentiate it from a standard `.cpp` file. The `.tcc` file is explicitly included from within the header file.

As a concrete example, we refer to Appendix A.1 of this document, where we give a complete implementation of our Tally Sheet project. In particular, we give code for `TallySheet.h` in Figure 28. We draw particular attention to line 80 of that source code which reads

```
#include "TallySheet.tcc"
```

This causes all the definitions from the second file to be explicitly included from within the header, and thereby indirectly included by any other file that includes the header.

Finally, we look at the style for implementing member functions within the `TallySheet.tcc` file, as shown in Figure 29 in the appendix. As was the case with our Point class, it is important to re-establish the proper scope when declaring the function signature. Because we are defining functions of a templated class, we must use the formal template syntax for each individual function, as shown in the following excerpt.

```
template <typename T>
string TallySheet<T>::_makeLabel(int ind) const {
  stringstream converter;
  converter << T(ind + _minV);
  string output;
  converter >> output;
  return output;
}
```

12.6 Unit Testing

In Python, we demonstrated how unit tests could be embedded in the same file as a module using the construct

```
if __name__ == '__main__':
```

That convenient style allows for test code to be run conditionally when the interpreter is executed directly on that file, but ignored when the module is imported from some other file.

Unfortunately, there is no standard way to embed unit testing within the same source code files as the code that is to be tested. The issue is that any executable must begin with a call to a main routine, but when multiple files are compiled and linked, there must be precisely one main routine defined. This was precisely the problem we pointed out with our original, single-file implementation of the gcd program from Figure 1 on page 7. Because the main routine is defined in that file, it becomes impossible for any other applications with their own main routine to be combined with this definition of the gcd function. That is why the revised version introduced at the beginning of this section relies on having a separate `gcdTest.cpp` file for testing. The unit test can be compiled by combining both `gcd.cpp` and `gcdTest.cpp` when compiling. However, we could build other executables that rely on gcd by linking some source code with `gcd.cpp` (but without `gcdTest.cpp`).

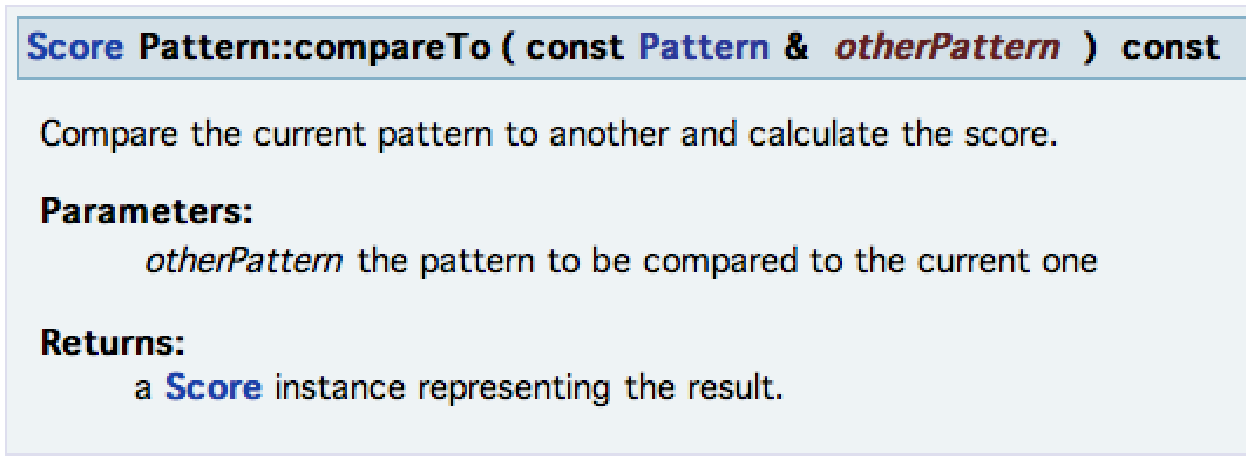
12.7 Documentation

Python also provided a built-in mechanism for embedding documentation strings directly within the source code files, with standard tools for generating documentation based upon those strings. Although comments can be embedded within C++ source code, there is no support for generating documentation from those comments in the standard C++ distribution. That said, there is a widely-used, third-party tool known as Doxygen (see `www.doxygen.org`) that can be used to embed actionable documentation within source code.

Doxygen supports several choices of conventions. One such style relies on comments between delimiters /** and */. Note that the starting delimiter /** begins with /* and therefore it is recognized by the C++ compiler as a standard comment. But the additional asterisk causes Doxygen to treat this as formal documentation. We demonstrate this style of comments throughout the source code given in the appendix to this document. As an example, Figure 35 of Appendix A.2 shows a documented version of a Pattern class for the game Mastermind. The compareTo function of that class is documented as follows:

```
/**
 * \brief Compare the current pattern to another and calculate the score.
 *
 * \param otherPattern the pattern to be compared to the current one
 * \return a Score instance representing the result.
 */
Score compareTo(const Pattern& otherPattern) const;
```

Within the comment, we use reserved control sequences such as \brief, \param, and \return to tag key pieces of information. When compiled with Doxygen, the documentation for the function might be rendered as in Figure 27.



Score Pattern::compareTo (const Pattern & otherPattern) const

Compare the current pattern to another and calculate the score.

Parameters:
otherPattern the pattern to be compared to the current one

Returns:
a Score instance representing the result.

Figure 27: Sample documentation produced by Doxygen.

A Full Source Code

As a conclusion, we offer complete source code that mirrors several projects that were implemented in Python within our book.

A.1 Tally Sheet and Frequency Counting

This section contains complete C++ code for a project that closely mirrors the Python project originally presented in Chapter 8.5.3 of our book for computing frequency counts. The TallySheet class we use here is similar to the templated one from Section 9.2, however this time we break it into separate `TallySheet.h` and `TallySheet.cpp` files, shown in Figures 28 and 29 respectively, including full documentation. We also include a C++ version of the FileUtilities component given in our Python version. This consists of files `FileUtilities.h` and `FileUtilities.cpp`, Figures 30 and 31 respectively. Finally, we give two sample applications using these tools. Specifically, Figure 32 displays the contents of file `CountLetters.cpp`, the main driver for computing letter frequencies, and Figure 33 contains `CountScores.cpp`, the corresponding driver for counting integer scores.

A.2 Mastermind

As our final project, we implement a text-based version of the Mastermind game, based closely upon the design of our Python version from Chapter 7 of our book. This project consists of the following components:

- a Score class as defined in `Score.h` of Figure 34. There is no `Score.cpp` because we in-line all of the function bodies for this simple class.
- a Pattern class as defined in `Pattern.h` of Figure 35 and `Pattern.cpp` of Figure 36.
- a TextInput class as defined in `TextInput.h` of Figure 37 and `TextInput.cpp` of Figure 38.
- a TextOutput class as defined in `TextOutput.h` of Figure 39 and `TextOutput.cpp` of Figure 40.
- a Mastermind class as defined in `Mastermind.h` of Figure 41 and `Mastermind.cpp` of Figure 42.
- The main driver for our text-based game, implemented in `Mastermind_main.cpp` of Figure 43.

```
#ifndef TALLYSHEET_H
#define TALLYSHEET_H
#include <iostream>

/**
 * \brief Manage tallies for a collection of values.
 *
 * Values can either be from a consecutive range of integers, or a
 * consecutive sequence of characters from the alphabet.
 */
template <typename T>
class TallySheet {
private:
  T _minV;
  T _maxV;
  int _size;
  int* _tallies;

public:
  /**
   * \brief Create an initially empty tally sheet.
   *
   * \param minVal the minimum acceptable value for later insertion
   * \param maxVal the minimum acceptable value for later insertion
   */
  TallySheet(const T& minVal, const T& maxVal);

  /**
   * \brief Make a copy of the given instance.
   */
  TallySheet(const TallySheet<T>& other);

  /**
   * \brief Assign one instance the state of another.
   */
  TallySheet<T>& operator=(const TallySheet<T>& other);

  /**
   * \brief Destruct the current instance.
   */
  ~TallySheet( );
```

Figure 28: Contents of `TallySheet.h` file. (continued on next page).

```
  /**
   *  \brief Increment the tally for the respective value.
   */
  void increment(const T& val);

  /**
   *  \brief Return the total number of current tallies for the given value.
   */
  int getCount(const T& val) const;

  /**
   *  \brief Return the total number of current tallies.
   */
  int getTotalCount( ) const;

  /**
   *  \brief Write a comprehensive table of results.
   *
   *  Report each value, the count for that value, and the percentage usage.
   *
   *  \param out an open output stream.
   */
  void writeTable(std::ostream& out) const;

private:
  /**
   *  \brief Convert from a native value to a legitimate index.
   *
   *  \return the resulting index (such that _minV is mapped to 0)
   */
  int _toIndex(const T& val) const;

  /**
   *  \brief Convert index to a string in native range.
   */
  std::string _makeLabel(int ind) const;
};

#include "TallySheet.tcc"
#endif
```

Figure 28 (continuation): Contents of `TallySheet.h` file.

```
#include <iostream>
#include <stdexcept>
#include <iomanip>
#include <sstream>
using namespace std;

template <typename T>
TallySheet<T>::TallySheet(const T& minVal, const T& maxVal) :
        _minV(minVal), _maxV(maxVal), _size(maxVal−minVal+1), _tallies(new int[_size]) {
  for (int i=0; i < _size; i++)
    _tallies[i] = 0;
}

template <typename T>
TallySheet<T>::TallySheet(const TallySheet<T>& other) :
      _minV(other._minV), _maxV(other._maxV), _size(other._size), _tallies(new int[_size]) {
  for (int i=0; i < _size; i++)
    _tallies[i] = other._tallies[i];
}

template <typename T>
TallySheet<T>::~TallySheet( ) {
  delete[ ] _tallies;
}

template <typename T>
TallySheet<T>& TallySheet<T>::operator=(const TallySheet<T>& other) {
  if (this != &other) {            // ignore self−assignments
    _minV = other._minV;
    _maxV = other._maxV;
    _size = other._size;
    delete[ ] _tallies;            // throw away old array
    _tallies = new int[_size];     // create new array
      for (int i=0; i < _size; i++)
        _tallies[i] = other._tallies[i];
  }
  return *this;
}

template <typename T>
void TallySheet<T>::increment(const T& val) {
  int ind = _toIndex(val);
  if (!(0 <= ind && ind <= _size))
    throw range_error("Parameter out of range");
  _tallies[ind] += 1;
}
```

Figure 29: Contents of `Tallysheet.tcc` file. (continued on next page).

```
template <typename T>
int TallySheet<T>::getCount(const T& val) const {
  int ind = _toIndex(val);
  if (!(0 <= ind && ind <= _size))
    throw range_error("Parameter out of range");
  return _tallies[ind];
}

template <typename T>
int TallySheet<T>::getTotalCount( ) const {
  int sum = 0;
  for (int i=0; i < _size; i++)
    sum += _tallies[i];
  return sum;
}

template <typename T>
void TallySheet<T>::writeTable(ostream& out) const {
  out << "Value  Count Percent \n----- ------ -------\n";
  int total = getTotalCount( );
  if (total == 0)
    total = 1;                    // avoid division by zero

  for (int ind=0; ind < _size; ind++) {
    string label = _makeLabel(ind);
    int count = _tallies[ind];
    float pct = 100.0 * count / total;
    out << setw(5) << label << setw(7) << count;
    out << fixed << setw(7) << setprecision(2) << pct << endl;
  }
}

template <typename T>
int TallySheet<T>::_toIndex(const T& val) const {
  int i = val − _minV;
  return i;
}

template <typename T>
string TallySheet<T>::_makeLabel(int ind) const {
  stringstream converter;
  converter << T(ind + _minV);
  string output;
  converter >> output;
  return output;
}
```

Figure 29 (continuation): Contents of `Tallysheet.tcc` file.

```
#ifndef FILE_UTILITIES_H
#define FILE_UTILITIES_H
#include <string>
#include <fstream>

/**
 * \brief Repeatedly prompt user for filename until successfully opening with read access.
 *
 * \param fin input file stream to be opened
 */
void openFileReadRobust(std::ifstream& fin);

/**
 * \brief Repeatedly prompt user for filename until successfully opening with write access.
 *
 * \param fout output file stream to be opened
 * \param defaultName a suggested filename.   This will be offered
 *           within the prompt and used when the return key is pressed without
 *           specifying another name
 */
void openFileWriteRobust(std::ofstream& fout, const std::string& defaultName);
#endif
```

Figure 30: `FileUtilities.h` header file.

```
#include "FileUtilities.h"
#include <fstream>
#include <iostream>
using namespace std;

void openFileReadRobust(ifstream& source) {
  source.close( );                                  // disregard any previous usage of the stream
  while (!source.is_open( )) {
    string filename;
    cout << "What is the filename? ";
    getline(cin, filename);
    source.open(filename.c_str( ));
    if (!source.is_open( ))
      cout << "Sorry. Unable to open file " << filename << endl;
  }
}
```

Figure 31: `FileUtilities.cpp` implementation (continued on next page).

```
void openFileWriteRobust(ofstream& fout, const string& defaultName) {
  cout << "Within openFileWriteRobust" << endl;
  fout.close( );
  while (!fout.is_open( )) {
    string filename;
    cout << "What should the output be named [" << defaultName << "]? ";
    getline(cin, filename);
    if (filename.size( ) == 0)
      filename = defaultName;
    fout.open(filename.c_str( ));
    if (!fout.is_open( ))
      cout << "Sorry. Unable to write to file " << filename << endl;
  }
}
```

Figure 31 (continuation): `FileUtilities.cpp` implementation..

```
#include "FileUtilities.h"
#include "TallySheet.h"
#include <cctype>                // provides isalpha and toupper
#include <iostream>
#include <fstream>
using namespace std;

int main( ) {
  ifstream source;
  ofstream tallyfile;
  TallySheet<char> sheet('A', 'Z');
  cout << "This program counts the frequency of letters." << endl;
  cout << "Only alphabetic characters are considered." << endl << endl;
  openFileReadRobust(source);
  while (!source.eof( )) {
    char character;
    source >> character;
    if (isalpha(character))
      sheet.increment(toupper(character));
  }
  source.close( );

  openFileWriteRobust(tallyfile, "frequencies.txt");
  sheet.writeTable(tallyfile);
  tallyfile.close( );
  return 0;
}
```

Figure 32: Main driver for computing frequency of letter usage.

```cpp
#include "FileUtilities.h"
#include "TallySheet.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

int main( ) {
  ifstream source;
  ofstream tallyfile;

  cout << "This program tallies a set of integer scores." << endl;
  cout << "There should be one integer per line." << endl << endl;

  openFileReadRobust(source);
  vector<int> values;
  while (source.good( )) {
    int val;
    source >> val;
    if (source.good( ))
      values.push_back(val);
    else {                              // ignore noninteger line
      source.clear( );
      source.ignore(std::numeric_limits<int>::max( ), '\n');
    }
  }
  source.close( );

  int small, large;
  small = *min_element(values.begin( ), values.end( ));
  large = *max_element(values.begin( ), values.end( ));
  TallySheet<int> sheet(small, large);
  for (int i=0; i < values.size( ); i++)
    sheet.increment(values[i]);

  openFileWriteRobust(tallyfile, "frequencies.txt");
  sheet.writeTable(tallyfile);
  tallyfile.close( );
  cout << "The tally has been written." << endl;

  return 0;
}
```

Figure 33: Main driver for computing frequency of integer scores.

```
#ifndef SCORE_H
#define SCORE_H

/**
 * \brief A score for a single turn from game of Mastermind.
 *
 * A "black" component designates the number of pegs that are
 * exact matches for the answer.   A "white" component counts
 * pegs that are correctly colored but not well positioned.
 */
class Score {
private:
  int _numBlack;
  int _numWhite;

public:
  /**
   * \brief Create score with given black and white components.
   *
   * \param numBlack the black component of the score
   * \param white the white component of the score
   */
  Score(const int numBlack, const int numWhite)
    : _numBlack(numBlack), _numWhite(numWhite) { }     // in-lined implementation

  /**
   * \brief Get the black component of the score.
   *
   * \return the number of pegs scored as black
   */
  int getNumBlack( ) const {
    return _numBlack;                    // in-lined implementation
  }

  /**
   * \brief Get the white component of the score.
   *
   * \return the number of pegs scored as white
   */
  int getNumWhite( ) const {
    return _numWhite;                    // in-lined implementation
  }
};
#endif
```

Figure 34: Contents of `Score.h` file for Mastermind project. No `Score.cpp` is necessary since all function bodies are in-lined.

```
#ifndef PATTERN_H
#define PATTERN_H

#include "Score.h"
#include <vector>

/**
 *  \brief Class for storing a color pattern for Mastermind.
 */
class Pattern {
private:
  std::vector<int> _pegList;

public:
  /**
   *  \brief Construct a new pattern.
   *
   *  Initially, the pattern consists of numPegs pegs, each set to color 0.
   *
   *  \param numPegs the length of the pattern
   */
  Pattern(const int numPegs);

  /**
   *  \brief return the length of the current pattern
   *
   *  \return the length of the pattern
   */
  int len( ) const;

  /**
   *  \brief Return the current color setting (an integer) of the specified peg.
   *
   *  \param index the index of the peg
   *  \return the peg's color
   */
  int getPegColor(const int index) const;

  /**
   *  \brief Set the color of a peg at the given index of the pattern.
   *
   *  \param index the index of the peg
   *  \param colorID the desired color identifier (an integer)
   */
  void setPegColor(const int index, const int colorId);
```

Figure 35: Contents of `Pattern.h` file (continued on next page).

```
  /**
   *  \brief Compare the current pattern to another and calculate the score.
   *
   *  \param otherPattern the pattern to be compared to the current one
   *  \return a Score instance representing the result.
   */
  Score compareTo(const Pattern& otherPattern) const;

  /**
   *  \brief Make a random pattern.
   *
   *  \param numColors the maximum number of colors to use in the pattern
   */
  void randomize(const int numColors);
};

#endif
```

Figure 35 (continuation): Contents of `Pattern.h` file.

```
#include "Pattern.h"
#include <set>
#include <algorithm>          // includes the 'count' method
#include <stdlib.h>
using namespace std;

Pattern::Pattern(const int numPegs) : _pegList(numPegs) {
  int i;
  for (i=0; i < numPegs; i++)
    _pegList[i] = 0;
}

int Pattern::len( ) const {
  return _pegList.size( );
}

int Pattern::getPegColor(const int index) const {
  return _pegList[index];
}

void Pattern::setPegColor(const int index, const int colorId) {
  _pegList[index] = colorId;
}
```

Figure 36: Contents of `Pattern.cpp` file (continued on next page).

```
Score Pattern::compareTo(const Pattern& otherPattern) const {
  // First calculate the black component of the score
  int black = 0;
  int i;
  for (i=0; i<_pegList.size( ); i++) {
    if (getPegColor(i) == otherPattern.getPegColor(i)) {
      black++;
    }
  }

  // The white component is a little more difficult to calculate.
  // First find out the colors used in the current pattern
  set<int> colorsUsed;
  set<int>::iterator iter;
  for (i=0; i<_pegList.size( ); i++)
    if (colorsUsed.count(_pegList[i]) == 0)
      colorsUsed.insert(_pegList[i]);

  // For each color used find the smaller number of times
  // it appears in each pattern and add them up.
  int white = 0;
  set<int>::iterator colorIter;
  int color, count1, count2;
  for (colorIter=colorsUsed.begin( ); colorIter!=colorsUsed.end( ); ++colorIter) {
    color = *colorIter;
    count1 = count(_pegList.begin( ), _pegList.end( ), color);
    count2 = count(otherPattern._pegList.begin( ), otherPattern._pegList.end( ), color);
    if (count1 < count2)
      white += count1;
    else
      white += count2;
  }
  white -= black;                                     // Don't count pegs that are paired up.

  return Score(black, white);
}

void Pattern::randomize(const int numColors) {
  int i;
  for (i=0; i<_pegList.size( ); i++)
    setPegColor(i, rand( ) % numColors);
}
```

Figure 36 (continuation): Contents of `Pattern.cpp` file.

```cpp
#ifndef TEXTINPUT_H
#define TEXTINPUT_H

#include "Pattern.h"
#include <string>
#include <vector>
using std::string;
using std::vector;

/**
 * \brief Class for dealing with text-based input for the Mastermind game.
 */
class TextInput {
private:
  int _lengthOfPattern;
  int _numColorsInUse;
  string _palette;

public:
  /**
   * \brief Create a new text input instance.
   *
   * \param colorNames  a list of strings (each color must start with a different letter)
   */
  TextInput(const vector<string>& colorNames);

  /**
   * \brief Ask the user how many pegs in the secret pattern.
   *
   * The length of the pattern is also stored internally.
   *
   * \return the length of the pattern
   */
  int queryLengthOfPattern( );

  /**
   * \brief Ask the user how many colors to use for secret pattern.
   *
   * The number of colors is also stored internally.
   *
   * \return the number of colors
   */
  int queryNumberOfColors( );
```

Figure 37: Contents of `TextInput.h` file (continued on next page).

```
  /**
   * \brief Ask the user maximum number of guesses to be allowed.
   *
   * \return the maximum number of guesses
   */
  int queryNumberOfTurns( ) const;

  /**
   * \brief Offer the user a new game..
   *
   * \return true if accepted, false otherwise
   */
  bool queryNewGame( ) const;

  /**
   * \brief Get a guess from the user and return it as a Pattern instance.
   *
   * \return the pattern entered
   */
  Pattern enterGuess( ) const;

private:
  /**
   * \brief Robustly prompt the user for an integer from small to large.
   */
  int _readInt(const string& prompt, int small, int large) const;
};

#endif
```

Figure 37 (continuation): Contents of `TextInput.h` file.

```
#include "TextInput.h"
#include <iostream>
#include <sstream>
using namespace std;

TextInput::TextInput(const vector<string>& colorNames) :
      _lengthOfPattern(0), _numColorsInUse(0), _palette("") {
  vector<string>::const_iterator iter;
  for (iter=colorNames.begin( ); iter!=colorNames.end( ); ++iter)
    _palette.push_back((*iter)[0]);
}
```

Figure 38: Contents of `TextInput.cpp` file (continued on next page).

```
int TextInput::_readInt(const string& prompt, int small, int large) const {
  string buffer;
  int answer = small − 1;                                   // intentionally invalid
  while (!(small <= answer && answer <= large)) {
    cout << prompt << " (from " << small << " to " << large << ")? ";
    cin >> buffer;
    stringstream converter;
    converter << buffer;
    converter >> answer;
    if (!converter.fail( )) {
      if (!(small <= answer && answer <= large))
        cout << "Integer must be from " << small << " to " << large << "." << endl;
    }
    else {
        cout << "That is not a valid integer." << endl;
    }
  }
  return answer;
}

int TextInput::queryLengthOfPattern( ) {
  _lengthOfPattern = _readInt("How many pegs are in the secret", 1, 10);
  return _lengthOfPattern;
}

int TextInput::queryNumberOfColors( ) {
  _numColorsInUse = _readInt("How many colors are available", 2, _palette.size( ));
  return _numColorsInUse;
}

int TextInput::queryNumberOfTurns( ) const {
  return _readInt("How many turns are allowed", 1, 20);
}

bool TextInput::queryNewGame( ) const {
  cout << endl;
  cout << "Would you like to play again? " << endl;
  string answer;
  cin >> answer;
  return (answer == "y"  answer == "Y"  answer == "yes"
    answer == "YES"  answer == "YES");
}
```

Figure 38 (continuation): Contents of `TextInput.cpp` file (continued on next page).

```
Pattern TextInput::enterGuess( ) const {
  Pattern pattern(_lengthOfPattern);
  string currentPalette = _palette.substr(0,_numColorsInUse);
  string patternString;
  int i;
  bool validPattern = false;
  while (not validPattern) {
    cout << endl;
    cout << "Enter a guess (colors are ";
    cout << _palette.substr(0, _numColorsInUse) << "): ";
    cin >> patternString;

    validPattern = true;
    if (patternString.size( ) != _lengthOfPattern) {
      cout << "The pattern must have " << _lengthOfPattern << " pegs" << endl;
      validPattern = false;
    } else {
      for (i=0; i<_lengthOfPattern; i++)
        if (currentPalette.find(toupper(patternString[i])) > _lengthOfPattern)
          validPattern = false;
      if (!validPattern)
        cout << "The color options are " << currentPalette << endl;
    }

    if (validPattern) {
      for (i=0; i<_lengthOfPattern; i++)
        pattern.setPegColor(i, _palette.find(toupper(patternString[i])));
    }
  }

  return pattern;
}
```

Figure 38 (continuation): Contents of TextInput.cpp file.

```
#ifndef TEXTOUTPUT_H
#define TEXTOUTPUT_H

#include "Pattern.h"
#include "Score.h"
#include <string>
#include <vector>
using std::string;
using std::vector;
```

Figure 39: Contents of TextOutput.h file (continued on next page).

```
/**
 * \brief Provide text−based output for the Mastermind game.
 */
class TextOutput {
private:
  string _colorOptions;
  int _currentTurnNum;
  int _lengthOfPattern;
  int _maxNumberOfTurns;

public:
  /**
   * \brief Construct a new TextOutput instance.
   *
   * \param colorNames a sequence of strings (each color must start with a different letter)
   */
  TextOutput(const vector<string>& colorNames);

  /**
   * \brief Game is beginning with specified parameters.
   */
  void startGame(int lengthOfPattern, int maxNumberOfTurns);

  /**
   * \brief Display recent guess Pattern and resulting Score to the screen.
   */
  void displayTurn(const Pattern& guess, const Score& result);

  /**
   * \brief Inform the player that he/she has correctly matched the secret Pattern.
   */
  void announceVictory(const Pattern& secret) const;

  /**
   * \brief Inform the player that he/she has lost and reveal the secret Pattern.
   */
  void announceDefeat(const Pattern& secret) const;

private:
  /**
   * \brief Returns string representation of given Pattern using color shorthands.
   */
  string _patternAsString(const Pattern& thePattern) const;
};

#endif
```

Figure 39 (continuation): Contents of `TextOutput.h` file.

```
#include "TextOutput.h"
#include <iostream>
using namespace std;

TextOutput::TextOutput(const vector<string>& colorNames) :
    _currentTurnNum(0), _lengthOfPattern(0), _maxNumberOfTurns(0), _colorOptions("") {
  vector<string>::const_iterator iter;
  for (iter=colorNames.begin( ); iter!=colorNames.end( ); ++iter)
    _colorOptions.push_back((*iter)[0]);
}

void TextOutput::startGame(int lengthOfPattern, int maxNumberOfTurns) {
  _currentTurnNum = 0;
  _lengthOfPattern = lengthOfPattern;
  _maxNumberOfTurns = maxNumberOfTurns;
}

void TextOutput::displayTurn(const Pattern& guess, const Score& result) {
  _currentTurnNum++;
  cout << "On turn " << _currentTurnNum << " of " << _maxNumberOfTurns
       << " guess " << _patternAsString(guess) << " scored "
       << result.getNumBlack( ) << " black and " << result.getNumWhite( )
       << " white." << endl;
}

void TextOutput::announceVictory(const Pattern& secret) const {
    cout << endl;
    cout << "Congratulations, you won!" << endl;
    cout << "The secret was " << _patternAsString(secret) << endl;
}

void TextOutput::announceDefeat(const Pattern& secret) const {
    cout << endl;
    cout << "The secret was " << _patternAsString(secret) << endl;
    cout << "Good luck next time." << endl;
}

string TextOutput:: _patternAsString(const Pattern& thePattern) const {
  string display;
  int i;
  for (i=0; i<_lengthOfPattern; i++)
    display.push_back(_colorOptions[thePattern.getPegColor(i)]);
  return display;
}
```

Figure 40: Contents of `TextOutput.cpp` file.

```
#ifndef MASTERMIND_H
#define MASTERMIND_H

#include "TextInput.h"
#include "TextOutput.h"

/**
 * \brief Main class for the Mastermind game.
 */
class Mastermind {
private:
  TextInput& _inputManager;
  TextOutput& _outputManager;

public:
  /**
   * \brief Create a new instance of the Mastermind game.
   *
   * \param inputManager instance of class that gathers input from the user
   * \param outputManager instance of class that displays output to the user
   */
  Mastermind(TextInput& inputManager, TextOutput& outputManager);

private:
  /**
   * \brief Play one game.
   */
  void _runSingleGame( );
};

#endif
```

Figure 41: Contents of `Mastermind.h` file.

```
#include "Mastermind.h"

Mastermind::Mastermind(TextInput& inputManager, TextOutput& outputManager) :
    _inputManager(inputManager), _outputManager(outputManager) {
  bool playAgain = true;
  while (playAgain) {
    _runSingleGame( );
    playAgain = _inputManager.queryNewGame( );
  }
}

void Mastermind::_runSingleGame( ) {
  // get parameters from the user
  int lengthOfPattern = _inputManager.queryLengthOfPattern( );
  int numberOfColors = _inputManager.queryNumberOfColors( );
  int maxNumberOfTurns = _inputManager.queryNumberOfTurns( );
  _outputManager.startGame(lengthOfPattern, maxNumberOfTurns);

  // pick a new secret
  Pattern secret(lengthOfPattern);
  secret.randomize(numberOfColors);

  // start playing
  int round = 0;
  Pattern guess(lengthOfPattern);
  bool victory = false;
  while (round < maxNumberOfTurns && !victory) {
    round++;
    // enact a single turn
    guess = _inputManager.enterGuess( );
    Score result = guess.compareTo(secret);
    _outputManager.displayTurn(guess, result);
    if (result.getNumBlack( ) == lengthOfPattern)
      victory = true;
  }

  if (victory)
    _outputManager.announceVictory(secret);
  else
    _outputManager.announceDefeat(secret);
}
```

Figure 42: Contents of `Mastermind.cpp` file.

```
#include "Mastermind.h"

int main( ) {
  vector<string> palette;
  palette.push_back("Red");
  palette.push_back("Blue");
  palette.push_back("Green");
  palette.push_back("White");
  palette.push_back("Yellow");
  palette.push_back("Orange");
  palette.push_back("Purple");
  palette.push_back("Turquoise");

  TextInput input(palette);
  TextOutput output(palette);
  Mastermind game(input, output);

  return 0;
}
```

Figure 43: Contents of `Mastermind_main.cpp` file.