A Transition Guide: Python to C++

Michael H. Goldwasser David Letscher

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

The Purpose of This Guide

Python is a wonderful programming language and we expect that readers of this book will find many opportunities to use it. That said, there are many different programming languages used by software developers. Each language has its own strengths and weaknesses, and professionals must become accustomed to programming in different languages. Fortunately, once you have a solid foundation in one language it becomes easier to transition to another language. This guide is designed for readers choosing C++ as a second language. C++ is among the most widely used languages in industry. As object-oriented languages, they have a great deal 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 that we have gained in Python. We begin in Section 1 by providing a high-level discussion about programming languages and the more significant differences between Python, and C++. Section 2 provides our first direct comparison between Python source code and C++ code. As much as possible, we rely upon earlier examples in Python and then translate these to C++. From there we will go into specific syntax and usage of C++. In Section 3 we discuss the major features of C++. Writing classes are discussed in Section 4 with coverage of Python and C++'s object models and memory management in Section 5. Storage of C++ objects in containers is discussed in Section **??**. Good software practices and working with larger programs, including the Mastermind case study is in Section **??**. And writing template based classes is in Section **??**. And in Section **??** we discuss how to integrate the use of Python and C++ in a single program. The full source code for all of the programs is contained in an appendix.

With that said, let the tour begin...

Contents

1 High-Level Programming Languages

To begin, we recommend that you reread Section 1.3 of the book, where we first describe 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. In fact, this is often done for specialized applications where 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.

1.1 Convenience versus Efficiency

In effect, high-level languages offer convenience. They support a greater range of data types and a richer syntax for expressing a series of operations. Yet this additional support is somewhat artificial. In the end, the software must be translated back to the CPU's machine language in order to be executed on a computer. For high-level languages, this translation has been automated in the form of a compiler or interpreter. As a result, software written in a high-level language is no more powerful than equivalent software that could have been written directly in the low-level language (given enough time and expertise). The convenience afforded by a high-level language often comes at the expense of some slight inefficiencies in the performance of the resulting software. The automated translation from high level to low level has been carefully optimized, but still the generated low-level code is not always as streamlined as code crafted directly by an expert in the field.

Yet as a society, we simply cannot afford to have each and every piece of software hand-crafted in a low-level language. While there are slight discrepancies in efficiency, those are quickly negated by improvements in hardware, networks, and other aspects of a computing environment. A more significant concern is the software development time, that is, the time it takes to carry an idea from the initial inspiration to the final software for a consumer. The design and development of quality software applications is extremely labor-intensive, and can take months or years depending on the project. The single biggest factor in the cost of a software project is employing the developers. So there is great benefit in use of a high-level language that can better support abstractions and thereby reduce the overall development cycle.

More than a thousand high-level languages have been developed over time, with perhaps a hundred that are still actively used for program development. What makes each language unique is the way in which concepts are abstracted and expressed. No single language is perfect, and each strikes its own balance in trying to support the development of efficient, maintainable, and reusable software. This guide is limited to the examination of three specific *object-oriented* languages, yet the object-oriented paradigm is just one example of an abstraction for program development. Even within the object-oriented framework, there are differences between languages. In the remainder of this section, we discuss the most significant ways in which Python and C++ differ.

1.2 Interpreter versus Compiler

An important aspect of any high-level language is the process by which it is translated back to the low-level machine code to be executed. 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. It adapts its outward behavior to match the semantics indicated by the given source code. In effect, the translation from the high-level code to low-level operations is performed on-the-fly, each time the program is run.

In contrast, C++ is an example of a *compiled* language. Progressing from the original source code to a running program is a two-step process. During the first phase ("compile-time"), the source code is fed as input to a special piece of software known as a *compiler*. That 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 another file known as an executable. During the second phase (the "run-time"), the executable is independently started by the user; the compiler is no longer needed unless a new executable must be generated, for example when a change is made to the original source code.

The greatest advantage of the compilation model is execution speed. In essence, the more that can be handled at compile-time, the less work there is to be done 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 second advantage is that the executable can be distributed to customers as free-standing software. So long as it was designed for the particular machine code of their system, it can be executed by those users without any further requirement of software installations (e.g., an interpreter). A consequence of this model is that the machine code can be distributed by a company without exposing the original source code that was used to generate it (although some companies choose to "open source" their software).

In contrast, there is no distinction between compile-time and run-time for a purely interpreted program. The interpreter bears the burden of translating the original source code as part of the run-time process. Furthermore, distributing the source code is only useful to a customer who has a compatible interpreter installed on his or her system. The primary advantage of an interpreted language is greater platform-independence. The same source code can be distributed for use on different computing platforms, so long as each platform has a valid interpreter. In contrast, a compiled executable is catered to one particular machine language; different versions of the executable must be distributed for use on different computing platforms.

For software developers, the debugging cycle varies a great deal when working with an interpreter language versus a compiled language. For example, we have readily used Python's interpreter not just as a means for running a final version of a program, but to provide useful feedback and interaction when problems arise. The compiler can be helpful in detecting purely syntactical errors at compile-time, but it is no longer of use when run-time errors occur.

The type of object associated with the identifier identifies what we can / cannot do with it. The interpreter determines this on the fly, not the compiler.

1.3 Dynamic versus Static Typing

For compiled languages, there is an advantage in [illegible] 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 known as a *dynamically typed* language. Within a given scope an identifier can be assigned to an underlying value using an assignment statement, as in

```
age = 38
```

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. In fact, we could later reassign that same identifier to the string `'Stone'`. Types are not formally associated with the identifiers, but rather with the underlying objects (thus the value 38 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 illegal if `age` is a string. The method call `age.lower( )` will be legitimate 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[1] whether the object currently associated with the name `age` supports the syntax `lower( )`. If so, the expression is evaluated successfully; if not, a runtime 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. The identifiers are assigned to the objects sent by the caller. The dynamic typing also applies to the attributes within a class definition, which are generally initialized in the constructor, but never explicitly declared.

In general, code works so long as the objects support the expected members; otherwise an exception is raised. 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. Another form of polymorphism is a function that displays markedly different behaviors depending upon the parameter type. For example in Section 6.2 of the book we provided a `Point.__mul__` implementation that used explicit type checking at run-time to determine the appropriate semantics for the use of multiplication.

C++ is statically typed languages. An explicit type declaration is required for every identifier before it can be used. The following demonstrates a type declaration followed by an assignment, as it might appear in C++:

```
int age;
age = 38;
```

The first line is a declaration that establishes the identifier `age` as an integer value in the current scope. Type declarations apply in many contexts. For example, a function signature must include explicit type declarations for all formal parameters, as well as for the resulting return type. All data members must be explicitly typed as part of a class definition. 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 = 38` 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 expected a floating-point number as in `sqrt("Hello")`, this error can be detected at compile-time. In some scenarios, type declarations can 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, yet this early detection is more significant for a compiled language, for which there

[1]See Section 12.6 of the book for a much more detailed explanation of the name resolution process in Python.

Python

```python
def gcd(u, v):
  # we will use Euclid's algorithm
  # for computing the GCD
  while v != 0:
    r = u % v     # compute remainder
    u = v
    v = r
  return u
```

C++

```cpp
  while (v != 0) {
    r = u % v;    // compute remainder
    u = v;
    v = r;
  }
  return u;
}

int main( ) {
  int a, b;
  cout << "Enter first number: ";
```

Figure 1: A function for computing the greatest common denominator, as seen in Python, and C++.

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 is the reduced syntactical burden associated with explicit declarations, together with the simpler support for polymorphism.

1.4 Why C++

In comparison to other object-oriented languages, the greatest strength of C++ is its potential for creating fast executables and robust libraries. This efficiency and power stems from a variety of factors. C and C++ provide great flexibility in controlling many of the underlying mechanism 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 wisely, this control can lead to a more streamlined result. Furthermore, because of the long history of C and C++ and their widespread use, the compiler 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. As a 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, and possibly flawed, software.

2 A First Glance at C++

To warm up, we begin with a side-by-side example of a Python code fragment and the corresponding code in C++. In particular, Figure 1 demonstrates the code for a function that computes the greatest common denominator of two integers using Euclid's algorithm. The Python version is similar to the solution for Practice 5.19, given on page 631 of the book.

Looking more carefully at Figure 1, we see that the C++ version is a bit more bulky than the Python code. It should hopefully seem legible, yet there are definitely syntactical differences. First we draw attention to Python's use of whitespace versus C++'s use of punctuation for delimiting the basic syntactic structure of the code. An individual command in Python (e.g., u = v) is followed by a newline character, officially designating the end of that command. In C++, each individual command must be explicitly terminated with a semicolon. For example, we see the semicolon after the command u = v on line 7.

There is also a difference in designating a “block” of code. In Python, each block is preceded by a colon, and then indentation is used to clearly designate the extent of the block. We see the body of a while loop consisting of lines 5–7 of the Python code, and that loop nested within the larger body of the gcd function, comprised of lines 2–8. 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 5 until the matching right brace on line 9. That loop is itself nested within the function body that begins with the left brace on line 1 and concludes with the right brace on line 11.

There is a difference in punctuation, as C++ require that the boolean condition for the while loop on line 5 be expressed within parentheses; we did not do so in Python, although parentheses could be used optionally. We also see a difference in the punctuation usage when providing inlined comments.[2] In Python, the # character is used to designate the remainder of the line as a comment, as seen at lines 2, 3 and 5. Two different comment styles are allowed in C++. A single-line comment is supported, though using the // pattern, as seen at line 6. Another style is demonstrated on lines 2 and 3, starting with the /* pattern, and ending with the */ pattern. This style is particularly convenient as it can span multiple lines of source code.

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, that is not formally part of the language syntax. The identical function could technically be defined in a single line as follows:

```
int main( ) {
```

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.

The more significant differences between the Python and C++ versions of our example involve the difference between ***dynamic*** and ***static*** typing, as we originally discussed in Section 1.3. Even in this simple example, there are three distinct manifestations of static typing. The formal parameters (i.e., identifiers u and v) are declared in the Python signature at line 1 without any explicit type designation. In the corresponding declaration of parameters in the C++ signature, we find explicit type declaration for each parameter with 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 our use of u and v within the function body. Second, it allows the compiler to enforce that integers be sent by the caller of our function.

The second manifestation of static typing is the explicit designation of the *return type* as part of a formal signature in C++. In line 1 of our C++ example, the declaration **int** at the beginning of the line labels this 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 10), 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, yet illegal if g has been declared as a string.

Finally, we note the declaration of variable r at line 4 of our C++ code. This designates r as a local variable representing an integer, allowing its use at lines 6 and 7. 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 has scope based upon the most specific set of enclosing braces at the point of its declaration. In our original example, the variable has scope as a local variable for the duration of the 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 this form of a restricted scope).

You may have noticed that this first example is not at all object oriented. In fact, the C++ code fragment

[2] We wait until later to visit the corresponding issue of embedding official documentation.

C++ Type	Description	Literals	Python analog
bool	logical value	**true** **false**	**bool**
short	integer (often 16 bits)		
int	integer (often 32 bits)	38	
long	integer (often 32 or 64 bits)	38L	**int**
			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: Most common primitive data types in C++.

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

that we give is essentially a C code fragment (the only aspect of the fragment which is not in accordance with the C language is the // style of comment at line 6).

3 C++ Fundamentals

3.1 Data Types and Operators

Figure 2 provides a summary of primitive data types in C++, noting the correspondence to Python's types. The **bool** type is supported by both languages, although the literals **true** and **false** are uncapitalized in C++ while capitalized in Python. C++ gives the programmer more fine-grained control in suggesting the underlying precision of integers, supporting three different fixed-precision integer types (**short**, **int**, and **long**). The precision of Python's **int** class is akin to C++'s **long**. Python's **long** type serves a completely different purpose, representing integers with unlimited magnitude. There is no such standard type in C++ (although some similar packages are independently available). Each of the integer types has an **unsigned** variant that represents only non-negative numbers. In order of size these types are **unsigned char**, **unsigned short**, **unsigned int**, and **unsigned long**. These should be using in situations were you know that the value will never be negative, e.g.,a index of a list.

C++ supports two different floating-point types (**float** and **double**) with a choice of underlying precisions. The **double** type in C++ is the more commonly used, and akin to what is named **float** in Python.

C++ also supports two different types for representing text. A **char** type provides a streamlined representation of a single character of text[3], 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.

In contrast with Python's immutable **str** class, A C++ string is *mutable*. A summary of the most commonly used string operations is given in Figure 3. Notice that the expression s[index] can be used to access a particular character, as well as to change that character to something else, when used on the left-hand side of an assignment. Also, the syntax s+t is used to generate a third string that is a concatenation of the others, while syntax s.append(t) mutates instance s.

[3]The **char** can also be used as an 8-bit numeric type with all of the same operations as the other integer types.

Non-mutating Behaviors	
s.size( ) or 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, start)	Returns the least index, greater than or equal to start, at which pattern begins; returns string::npos when not found.
s.rfind(pattern, start)	Returns the greatest index, less than or equal to indicated start, at which pattern begins; returns string::npos when not found.
s.find_first_of(charset, start)	Returns the least index, greater than or equal to indicated start, at which a character of the indicated string charset is found; returns string::npos when not found.
s.find_last_of(charset, start)	Returns the greatest index, less than or equal to indicated start, at which a character of the indicated string charset is found; returns string::npos when 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.

Mutating Behaviors	
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.insert(index, t)	Inserts copy of string t into string s at the given index.
s.insert(index, t, num)	Inserts num copies of t into string at the given index.
s.erase(start)	Removes all characters from start index 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 3: Selected behaviors supported by the string class in C++.

Type declarations

The basic form of a type declaration was demonstrated in our first glance of Section 2:

```
int r;
```

It is also possible to combine a variable declaration with an initial assignment statement, as in **int** age=38. However, the preferred syntax for initialization in C++ is the following:

```
int age(38);
```

Furthermore, we can declare multiple variables of the same type in a single statement (with or without initial values), as in

```
int age(38), zipcode(63103);        // two new variables
```

Immutability

Python often makes a distinction between mutable and immutable types (e.g., **list** vs. **tuple**). C++ takes a different approach. Types are generally mutable, yet a programmer can designate an individual instance as immutable. This is done by use of the **const** keyword as part of the declaration, as in

```
const int age(38);      // immortality
```

This immutability is strictly enforced by the compiler, so any subsequent attempt to change that value, as with age++, results in a compile-time error.

Operators

With a few notable discrepancies, the two languages support a very similar set of operators for the primitive types. The same basic logical operators are supported, yet C++ uses the following symbols (borrowed from the syntax of C):

&&	for logical **and**
||	for logical **or**
!	for logical **not**

For numeric values, Python differentiates between ***true division*** (i.e., /), ***integer division*** (i.e., //), and ***modular arithmetic*** (i.e., %), as originally discussed in Section 2.4 of the book. C++ supports operators / and %, but not // (in fact we already saw this pattern used to designate inline comments in C++). The semantics of the / 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 (a discussion of type conversion begins on the following page).

As is the case for Python, C++ supports an operator-with-assignment shorthand for most binary operators, as with x += 5 as a shorthand for x = x + 5. Furthermore, C++ supports a ++ 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. A −− operator is similarly supported in pre-decrement and post-decrement form. This combination of operators can be valuable for an experience programmer, but their use also leads to very subtle code and sometimes mistakes. We recommend that they be used sparingly until mastered.

Converting between types

In Python, we saw several examples 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 get an internal 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 lost of information. For example, saving a floating-point value into an integer variable causes any fractional portion to be truncated.

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

There are many 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, we can indicate an ***explicit cast***. This 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 b 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 such casting to perform all such 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, related to the handling of input and output, which we next discuss.

3.2 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, or 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 can insert data into a stream to send it elsewhere, or extract data from an existing stream. A stream that provides us with input is represented using the istream class, and a stream which we use to send output elsewhere is represented using the ostream class. Some streams (iostream) can serve as both input and output. Then there are more specific classes devoted to certain purposes (e.g., a file is managed with an fstream).

Necessary libraries

Technically, the definitions that we need are not automatically available in C++. Most are defined in a standard library named iostream (short for “input/output streams”). A C++ library serves a similar purpose to a Python module. We must formally include the definitions from the library before using them.

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

Technically, the first of these statements imports the library, while the second brings those definitions into our default namespace. In addition to the basic class definitions, that library defines two special instances for handling input to and from the standard console. cout (short for “console output”) is an object used to print messages to the user, and cin (short for “console input”) is used to get input typed by the user.

Console output

In C++ we generate console output through the cout stream from the iostream library. Streams support the operator << to insert data into the stream, as in cout << "Hello". The << symbol was chosen to subliminally suggest 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 in a single command by repeated use of the operator, as in cout << "Hello"<< " and "<< "Goodbye". 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 can directly embed the escape character, \n, within a string, C++ offers the more portable definition of a special object endl that represents a newline character.

To demonstrate several typical usage patterns, we provide the following side-by-side examples in Python and C++. In both cases, we assume that variables first and last have previously been defined as strings, and that count is an integer.

Python

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

C++

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

Formatted output

In Python, we use string formatting to more conveniently generate certain output, as in the following example:

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

The use of the % sign for this purpose is designed to mimic a long-standing routine named printf, which has been 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 usually the recommended approach for C++ (printf does not work consistently with the C++ string class). Instead, formatted output is generated directly through the output stream. Since data types are automatically converted to strings, the above example can be mimicked in C++ as

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

This approach is not quite as pleasing as the previous, but in this case, it does the job. More effort is needed to control other aspects of the formatting, such as the precision for floating-point values. In Python, the expression `'pi is %.3f'% 3.14159265` produces the result `'pi is 3.142'`. In C++, the iomanip library provides additional mechanisms for formatting output. These manipulators are sent to the output stream in the same way that endl is used. Common manipulators include setting the width of the next field using setw(width) or precision of the output. For example,

```
cout << setw(10) << 123 << endl;
```

would have display the number 123 right justified using 5 spaces. Note that this has no affect on any of the output after the number is displayed.

For floating point numbers, display can be either set to used fixed precision or scientific notation. And the precision can be set. To have all floats displayed in a using fixed precision with 2 digits after the decimal point try the following:

```
cout << fixed << setprecision(2);
```

And to use scientific notation use cout << scientific;. These formatting selections will stay in effect until changed. Other manipulators allow you to change justification, set the fill character, change the base numbers are displayed in, and control how **bool**'s are displayed.

Console input

Just as cout is an output stream for the console, cin is an input stream used to read from the console. Here is a simple example of its use.

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

The >> operator *extracts* data out from the stream and stores it in the given variable. The static typing of C++ actually helps us here, in comparison to Python. Since number was already clearly designated as an integer, C++ automatically converts the input as a number. Recall that in Python, we had to get the raw input string and explicitly convert it as in,

Python
```
number = int(raw_input('Enter a number from 1 to 10: '))
```

There are other differences regarding the treatment of input. In Python, the **raw_input** command reads one line at a time. In C++, extracting data from a stream uses up only as many characters as are necessary. For example, here is a code fragment that asks the user to enter two numbers on the same line and computes their sum.

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

For the sake of comparison, a Python version is given as the solution to Practice 2.31. To be fair, both of our versions crash if a user enters characters that cannot be interpreted as integers.

When a string is requested, the stream advances to the first nonwhitespace character and goes from there up until the next whitespace character. This is quite different than with Python. Assume that we execute name = **raw_input**('What is your name?), and the user responds as follows:

```
What is your name? John Doe
```

In Python, that entire line would be saved as a string 'John Doe'. If the same user interaction were given to the following C++ code,

```
string name;
cout << "What is your name? ";
cin >> name;
```

the result would be that name is assigned "John". The other characters remain on the stream until a subsequent extraction. If we want to read a full line at a time in C++, we can do so by using the syntax getline(cin, name).

File streams

File streams are defined in a library named fstream. They allow reading from and writing to files in a similar fashion to cin and cout. Everything we have discussed about input and output apply to file streams as well. There are three types of file streams available in C++: ifstream, ofstream and fstream. The first two are used for input only or output only, while fstream can be used for both input and output.

If the name of a file is known in advance, an associated file stream can be declared as

```
fstream datastream("data.txt");
```

If the filename is not known in advance, the stream can first be declared and then later opened, as

```
fstream datastream;
datastream.open("data.txt");
```

This syntax is typically used for opening an input file. The syntax which would correspond to Python's 'w' access mode is

```
fstream datastream("data.txt", ios::out);
```

Append mode can be specified by using ios::app in place of ios::out above.

String streams

A stringstream class, included from a sstream library, allows us to use our formatting tools to produce an internal string rather than external output. In essence, such a stream serves as a buffer, allowing us to use the stream operators to force data into the buffer, or extract data from the buffer. At the end of Section 3.1, we avoided the issue of how to convert an **int** to a string. This can be accomplished using a stringstream as follows:

```
stringstream temp;
temp << i;            // insert the integer representation into the stream
temp >> s;            // extract the resulting string out from the stream
```

3.3 Control Structures

We already saw an example of a **while** loop as part of our first example in Section 2. The basic structure is similar, with only 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. Here is a similar C++ code fragment for requesting a number between 1 and 10 until receiving such a number:

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

We should note that we have not properly handled the case when a noninteger is entered.

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 change a negative 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 a typical pattern does not require excessive braces. Our first example of a nested conditional in Python was given on page 144 of Section 4.4.2 of the book. That code could be written in C++ to mimic Python's indentation as follows (assuming groceries is an adequate container[4]):

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

[4]We will discuss C++ containers in Section B.5.8.

For loops

C++ supports a **for** loop, but with very different semantics that Python's. The style dates back to its existence in C. The original use was to provide a more legible form of the typical ***index-based*** loop pattern described in Section 4.1.1 of the 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!";
```

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. In fact, 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!";
```

The for loop is far more general. For example, 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;
```

3.4 Defining a Function

We already provided an example of a C++ function in our initial demonstration from Section 2. In this section we give a few other examples.

With that first example, we emphasized the need to explicitly designate the type of each individual parameter as well as the returned type. If the function does not provide a return value, there is a special keyword **void** that designates the lack of a type. Here is such a function that prints a countdown from 10 to 1.

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

We used an alternative version of this function in Section 5.2.2 of the 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;
}
```

3.5 Managing a Complete Program

We need to discuss one last detail before developing a complete C++ program. What should happen when the program is executed? Where does the flow of control begin? With Python, the flow of control begins at the beginning of the source code. The commands are interpreted one after another, and so a trivial Python program might appear simply as:

Python
```
print 'Hello World.'
```

In C++, statements cannot generally be executed without any context. When an executable is started by the operating system, the flow of control begins with a call to a special function named main. The above Python code is most directly translated into C++ as the following program:

```
#include <iostream>
using namespace std;
int main( ) {
  cout << "Hello World." << endl;
  return 0;
}
```

The return value for main is a bit of a technicality. The signature must designate an **int** return type. The actual value returned is reported back to the operating system at the conclusion of the program. It is up to the operating system as to how to interpret that value, although zero historically indicates a successful execution while other values are used as error codes.

To demonstrate how to compile and execute a C++ program, we introduce a second example in Figure 4. This is a rather simple program that pulls together several earlier techniques to compute and display the greatest common denominators of numbers entered by the user. Let's assume that the source code is saved in a file `gcd.cpp`. The most widely used compiler is from an organization known as GNU, and the compiler is most typically installed on a system as a program named `g++`. The compiler can be directly invoked from the operating system command line with the syntax,

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

The compiler will report any syntax errors that it finds, but if all goes well it produces a new file named `gcd` that is a true executable. It can be started on the computer just as you would start any other executable (with the Windows operating system, that executable might need to be named `gcd.exe`). There also exist integrated development environments for C++ (such as Python's IDLE). Those typically rely upon the same underlying compiler, but provide a more interactive control for the process.

The process is more involved when the software combines source code from multiple files (as is typical for larger applications). Our supplemental website demonstrates two such large projects, the text-based Mastermind program from Chapter 7 of the book, and the frequency-counting case study from Section 8.5.3 of the book based on the design of the TallySheet class.

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

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

int main( ) {
  int a, b;
  cout << "Enter first number: ";
  cin >> a;
  cout << "Enter second number: ";
  cin >> b;
  cout << "The gcd is " << gcd(a,b) << endl;
  return 0;
}
```

Figure 4: A complete C++ program that computes the gcd of two numbers.

4 Classes in C++

4.1 Using Classes

4.2 Defining a Class

To demonstrate the syntax for a C++ class, we rely upon several of the examples that we first used in Chapter 6 of the book, beginning 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 5. There are several important aspects to discuss in comparing C++'s syntax to Python's. Please bear with us as we highlight the key differences.

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 type of 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**).

```
 1  class Point {
 2    private:
 3      double _x;
 4      double _y;
 5
 6    public:
 7      Point( ) : _x(0), _y(0) { }
 8
 9      double getX( ) const {
10        return _x;
11      }
12
13      void setX(double val) {
14        _x = val;
15      }
16
17      double getY( ) const {
```

Figure 5: Implementation of a simple Point class.

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 particular case, having already initialized the two variables, there is nothing else for us to do. But the { } serves as a placeholder syntactically (somewhat like **pass** in Python).

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 9). For those who miss the self-reference, it is implicitly available in C++, although it is named **this**. This can be useful, for example, when passing our object as a parameter to an outside function.

Access control

Another distinction 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 Section 7.6 of the 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 rely upon a 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 subsequent 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 Python, 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 13. 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 declared 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 a class definition, we provide a more robust implementation of a Point class, modeled upon the Python version from Figure 6.4 on page 213. Our C++ version is shown in Figure 6.

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 lines 7 and 8.

Our next lesson is to re-emphasize that we can access members of the class without explicitly using a self-reference. This was already made clear in our use of names like _x rather than Python's **self**._x. The same convention is used when the body of one member function invokes another. At line 29, we see a call to the scale method. This is implicitly invoked on the current instance. Line 27 has a similar invocation of the distance method. Notice the use of Point( ) at line 27 to instantiate a new (default) point as a parameter to the distance function; this style is the same as we used in Python.

Lines 32–34 are used to support the + operator, allowing for the addition of two points. This behavior is akin to the __add__ method in Python, although in C++ the semantics are defined using **operator**+ as the "name" of a method. In the case of the syntax p + q, the point p technically serves as the implicit instance upon which this method is invoked, while q appears as a parameter in the signature. Technically, the **const** declaration that we make at line 32 designates that the state of p is unaffected by the behavior.

Lines 36–42 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 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.

Technically, line 43 of Figure 6 ends our Point class declaration. However, we provide two supporting definitions shown in Figure 7. The first of those supports use of a syntax such as 3 * p. The earlier definition of **operator*** from lines 36–38 technically supports the * operator when a Point instance is the *left-hand* operator (e.g., p * 3). C++ does not allow the formal class definition to affect the operator behavior in the case where the only instance of that class is the right-hand operator. Instead, we define such a behavior independent of the official class definition. So at lines 44–46, we provide a definition for how * should behave when the first operator 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 the 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).

Finally, lines 48–51 are used to produce a text representation of a point when inserted onto an output stream. A typical syntax for such a behavior is cout << p. Again we define this behavior outside of the

```
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;
    }

    void setX(double val) {
      _x = val;
    }

    double getY( ) const {
      return _y;
    }

    void setY(double val) {
      _y = val;
    }

    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( ) );
      if (mag > 0)
        scale(1/mag);
    }

    Point operator+(Point other) const {
```

Figure 6: Implementation of a robust Point class.

```
44    }
45
46    Point operator*(double factor) const {
47      return Point(_x * factor, _y * factor);
48    }
49
50    double operator*(Point other) const {
51      return _x * other._x + _y * other._y;
```

Figure 7: Supplemental operator definitions involving Point instances.

context of the class because the point serves as the right-hand operator. Line 49 inserts our desired output representation onto the given output stream. We use the formal parameter out rather than cout so that a user can apply this behavior to any type of output stream. The declared return type on line 48 and the return statement at line 50 are technically required to allow for multiple << operations to be applied on a single line; 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 is another technicality, leading to our next topic of discussion.

4.3 Inheritance

In Chapter 9 of the 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 that used a SortedSet. Although we omit the presumed definition for a basic Television class, our complete code for the DeluxeTV class is given in Figure 8. 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,[5] 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 of favorite channel numbers. Although we used our custom SortedSet class in our Python version, this is an excellent opportunity to make use of the C++ set container. 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 designing a class that inherits from another, the question arises as to whether code for that new class should be able to directly access members inherited from the parent. This is determined by the original access control designated for those members. A child class cannot access any members from the parent that are declared as **private**. However, the child is granted access to members designated as **protected** by the parent. 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;
  ...
```

[5]For the sake of simplicity, we will not discuss the precise significance of the term **public** on line 1.

```
class DeluxeTV : public Television {
 protected:
  set<int> _favorites;

 public:
  DeluxeTV( ) :
    Television( ),     // parent constructor
    _favorites( )      // empty set by default
    { }

  void addToFavorites( ) { if (_powerOn) _favorites.insert(_channel); }

  void removeFromFavorites( ) { if (_powerOn) _favorites.erase(_channel); }

  int jumpToFavorite( ) {
    if (_powerOn && _favorites.size( ) > 0) {
      set<int>::iterator result = _favorites.upper_bound(_channel);
      if (result == _favorites.end( ))
        result = _favorites.begin( );      // wrap around to smallest channel
      setChannel(*result);
    }
    return _channel;
  }
};  // end of DeluxeTV
```

Figure 8: Implementing a DeluxeTV class through inheritance.

```
 1  class Square : public Rectangle {
 2    public:
 3      Square(double size=10, Point center=Point( )) :
 4        Rectangle(size, size, center)        // parent constructor
 5        { }
 6
 7      void setHeight(double h) { setSize(h); }
 8      void setWidth(double w) { setSize(w); }
 9
10      void setSize(double size) {
11        Rectangle::setWidth(size);       // make sure to invoke PARENT version
12        Rectangle::setHeight(size);      // make sure to invoke PARENT version
13      }
14
15      double getSize( ) const { return getWidth( ); }
16  }; // end of Square
```

Figure 9: Implementing a Square class based upon a Rectangle.

If those had been declared as **private** we would not have the necessary access to implement our DeluxeTV. Of course, the original designer of the television may not have known that we would come along and want to inherit from it. However, an experienced C++ program 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.

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; then we take care of initializing the new favorites attribute. 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 a similar spirit, we choose to explicitly initialize the favorites attribute at line 8, although this too is superfluous, since we rely upon the default form of the constructor.

The rest of our DeluxeTV code is used to provide three new behaviors. The precise details of those methods depends greatly on our knowledge of the set class (which we admit we have not tried to explain). Our purpose for this example 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 make a call to the inherited method setChannel at line 20 of our code.

A Square class

As a second example of inheritance, Figure 9 provides a C++ rendition of our original Square class from Section 9.4.2 of the 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 do this we call to the parent constructor at line 4. In this case, we needed to do an explicit call to pass the appropriate dimensions and center. Had we not done this explicitly, an implicit call would have been made to the *default* version of the rectangle constructor, leading to incorrect

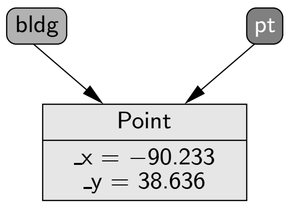

Figure 10: An example of parameter passing in Python.

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 use of the expression Rectangle:: before the method names at lines 11 and 12 designates our desire to use the parent version of those behaviors, rather than the default Square versions.

5 Assignments and the Object Model

Python presents 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 affect the passing of information to and from a function, as described in Section 10.3.1 of the 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:

Python
```python
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 10 diagrams the underlying configuration. This scenario is the precise result of the system performing an implicit assignment of formal parameter to actual parameter pt = bldg.

In this section, we consider the same issues in C++, namely the correspondence between an identifier and an underlying value, the semantics of an assignment statement, and the subsequent affect on passing information to and from a function. C++ provides more fine-tuned control than Python, allowing the programmer a choice between three different semantic models.

Value variables

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

```cpp
Point a;
```

causes the compiler to reserve memory for storing the state of a point. The translation from the name a to this particular instance is handled purely at *compile-time*, providing greater run-time efficiency than if that mapping were evaluated at run-time (as with Python). In the above case, the default constructor is applied, initializing both coordinates to zero. We could otherwise use an initialize statement to parameterize the

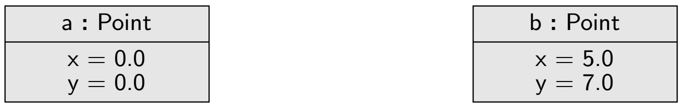

Figure 11: The declaration of two separate value variables.

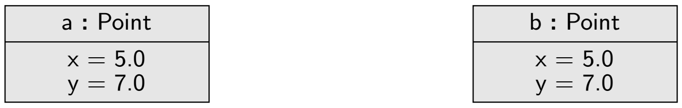

Figure 12: The effect of an assignment a = b upon value variables.

construction of a value variable, with a syntax such as the following.

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

Note that the parameters are enclosed in parentheses that follow the variable name (as opposed to the type name). Furthermore, note that we did not use any parentheses in the earlier case, when relying on the default constructor.

To portray the semantics of a value variable, we prefer a diagram in the style of Figure 11, without any independent concept of a reference. The assignment semantics for a value variable is very different from Python's. The assignment a = b causes the Point a to take on the *value* of Point b, as diagrammed in Figure 12. Notice that a and b are still names of two distinct points.

The difference between C++ and Python assignments has a notable effect on the semantics of a function. Going back to our earlier Python example, we might implement that function in C++ as follows:

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

The parameter passing model in this case is based on the implicit initialization of a formal parameter to the value of an actual parameter.

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

This initialization has very different consequences for a value variable. In this case, the formal parameter 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. Figure 13 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 ***by value***, as originally discussed on page 351.

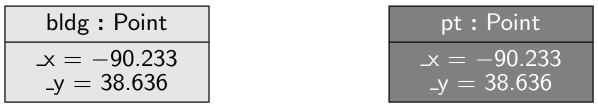

Figure 13: An example of passing by value in C++.

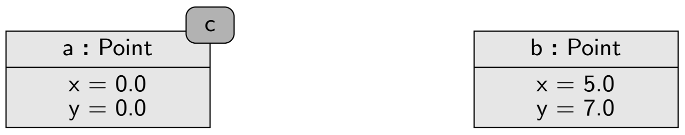

Figure 14: The name c is an example of a reference variable in C++.

Reference variables

The 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 14.

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 reassociated 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 the isOrigin function:

```
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, creating a copy is typically more expensive than creating an alias (which simply requires the communication of an underlying memory address). In fact, some classes do not even support the notion of creating a copy. For example, creating and manipulating a "copy" of an output stream is not allowed by the ostream class. For this reason, we see the use of references in the signature of **operator**<<, looking back at Figure 7; both the parameter stream and return value are communicated in such a way.

A second potential benefit of passing by reference is that it allows a function to intentionally manipulate the caller's object (in contrast to when a parameter is passed by value). In cases where a programmer wishes to get the efficiency of passing by reference, but without the risk of allowing the object to be mutated, the **const** modifier can be used in declaring a parameter, as in

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

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

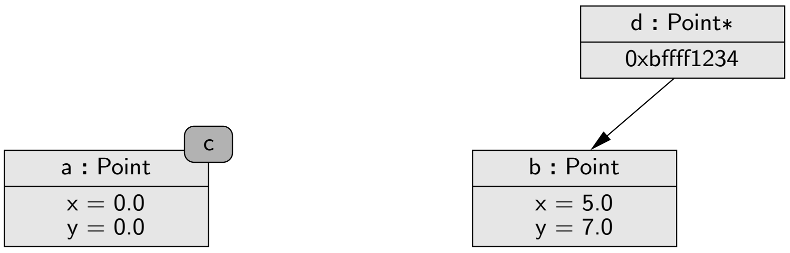

Figure 15: The variable d is an example of a pointer whose value is assigned to the address of instance b.

Pointer variables

C++ supports a third form of variable known as a ***pointer***. This provides a semantics which is closest to Python's model, although 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 d not to be a Point itself, but to be a variable that can store the memory address of a Point. This is related to a reference, but pointers are more general in that a pointer is allowed to point to nothing (using the keyword NULL in C++). Also, the pointer can be reassigned to different instances over time. A typical assignment statement is as follows:

```
d = &b;
```

This leads to a configuration diagrammed in Figure 15. 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. The expression *d is synonymous with b in this configuration. For example, we could call the method (*d).getY( ), which returns the value 7.0 in this case. However, since this syntax is bulky, C++ introduced a new operator in the context of pointers, allowing the syntax d->getY( ).

Passing a pointer provides a third alternative in the context of a function.

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

However, the only advantage to using a pointer variable rather than a reference variable is to allow for the possibility of sending a null pointer (recall that reference variables cannot have such a value).

Dynamic memory management

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

However, there are some circumstances when a program wants to take a more active role in controlling the underlying memory management. Pointer variables serve a valuable role in this regard. In our first example of a pointer variable, we assigned its value to the address of an *existing* object. We can also use an existing pointer to keep track of a brand new object, instantiated as follows:

```
d = new Point( );
```

This instantiation syntax is different from that of a value variable. The use of the keyword **new** designates this as what is termed a ***dynamic*** allocation. Memory will be set aside for storing and initializing this new Point, but the object is never automatically destroyed. It is up to the programmer to explicitly throw it away when it is no longer necessary, using a syntax **delete** d (just one more way that C++ allows a programmer to fine-tune software).