being specified in the implementation file also. We have shown this in our example, but commented out the incorrect lines and added the correct lines without the default values. The C++ compiler will give an error message if you forget to remove the default values from the implementation file when you cut and paste the function prototypes from the header file.

Since the code for these functions is short, the overhead of making the function call can take more execution time than the execution of the actual function code. C++ provides a mechanism known as *inline functions* to allow for more efficient execution. An inline function is generally written in a header file and is written exactly the same as the function would be in the implementation file except the keyword `inline` is placed before the function definition. In this case, the definition is also a declaration. For our conversion example, the header file with inline functions is

```
// conversions2.h

#ifndef __CONVERSIONS_H
#define __CONVERSIONS_H

inline double f_to_c(double f=0.0)
{
  return (f - 32.0) * (5.0 / 9.0);
}

inline double c_to_f(double c=0.0)
{
  return (9.0 / 5.0) * c + 32.0;
}

#endif
```

When writing all the functions inline in a header file, you do not need an implementation (`conversion.cpp`) file since all the information is contained in the header file. The `inline` keyword prevents multiple definitions of the file from being created when you link a number of different files that all include the `conversion2.h` header file.

If your inline function is relatively short, the compiler will generate the machine code for the function body and place it right in the code instead of creating the code to call a function. If your function is relatively long, the compiler will ignore your inline directive; instead it will create a normal function call since copying the machine code corresponding to a long function will make the program much larger if that function is called from a number of different places. A general rule is to declare functions that are less than five lines long as inline functions.

The original C programming language did not include inline functions and instead used preprocessor macros to accomplish the same result of not creating a function call for short functions. C++ also supports *macros* since they are part of the C language, but it is recommended that you use inline functions since they enforce type checking and are safer. A sample source file that defines and uses a macro for `c_to_f` is the following.

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

#define c_to_f(c) (9.0 / 5.0) * c + 32.0

int main()
{
  int x = 10;
  cout << c_to_f(x) << " ";
  cout << c_to_f(x + 10) << endl;
}
```

The `#define` preprocessor command is used to define macros. The preprocessor performs a search and replace for the item(s) inside the parentheses. Based on that, what do you think the output of this program is?

The two lines that use the macro are expanded by the preprocessor to

```
  cout << (9.0 / 5.0) * x + 32.0 << " ";
  cout << (9.0 / 5.0) * x + 10 + 32.0 << endl;
```

Given those expansions, you should now realize why the output of the program is `50 60`. The correct value for 20 degrees Celsius is 68 degrees Fahrenheit. You could fix this by using more parentheses in the macro, but there are still other potential problems with macros. So when writing C++ code, you should use the inline keyword instead of a macro to avoid the overhead of a function call.

There is an important issue regarding namespaces and header files. You should not use the version `using namespace ...` for any namespace in header files. The reason for this is that any file that includes your header file effectively would have that `using namespace` statement in it. This could cause problems if the source file defines a name that is also defined in the specified namespace. If you need to refer to a name defined within a namespace in a header file, always refer to it using the `namespace::name` syntax rather than including a `using` statement in your header file. We will see an example of this in section 9.4.

As Python contains a number of modules with useful functions, C++ has a standard library of functions. We have already seen the `iostream` header that the C++ language uses for input and output libraries. Many of the functions available in C++ are part of the original C standard library, but the header files have been updated for C++. The name of some of the C library header files are stdio.h, stdlib.h, and math.h. To use these header files in a C++ program, the .h extension is removed and the letter "c" is prepended, resulting in the names: cstdio, cstdlib, and cmath. For example, to use the `sqrt` function that is defined in the C math header file, you need the following statement at the top of your C++ file: `#include <cmath>`. There are other standard C++ header files, some of which will be covered later, but these along with `iostream` are the common ones beginning C++ programmers need.

The standard convention is to use less-than and greater-than signs around the names of header files that are part of the C++ library or libraries that are common and located in standard directories. Your C++ compiler also provides a method for specifying additional directories to search. On most systems the compiler first searches the additional directories you specify and then a set of standard directories containing header files. The first header file that matches the name is used. Double quotation marks must be used around header files that are in the same directory as the C++ source files you are compiling. For header files specified with double quotation marks, the compiler first searches the current directory. If the compiler cannot find the header file in the current directory, the compiler searches the additional user-specified include directories and the standard directories. You cannot use the less-than and greater-than signs around header files that are in the current directory since it is not searched by default, but you can use double quotation marks around standard header files since both the current directory and standard directories are searched. Even though it is possible to always use double quotation marks, C++ programmers follow the convention of using the less-than and greater-than signs for standard header files.

8.14 Assert Statements and Testing

Unlike Python which includes a unit testing framework, the C++ standard does not include a unit testing framework. There are a number of third-party, C++ unit testing frameworks that you can download and install. Most, if not all, of these frameworks are similar to Python's unit testing framework as both the C++ and Python unit testing frameworks are based on Java's unit testing framework. Instead

of covering one of the C++ unit testing frameworks, we will discuss the C++ `assert` statement since it allows you to easily write unit tests.

The Python unit testing framework provides a number of methods that verify if something is true and include "assert" as part of their name (e.g., `assertEquals` and `assertRaises`). These methods are based on the C++ `assert` statement (technically, it is a macro that the preprocessor expands) which takes a Boolean expression. If the Boolean expression is true, the program continues, but if it is false, the program exits immediately and indicates the line of code at which the `assert` statement fails. Unlike the Python unit testing framework which runs additional tests even after one of the tests fails, using the C++ `assert` statement causes your program to exit immediately if the assertion is not true. This means the tests following an `assert` statement that fails are not executed.

We will modify our `test_sort.cpp` file from section 8.13 to use the `assert` macro. The `assert` macro takes an expression and evaluates it. If the expression evaluates to true, execution continues. If the expression evaluates to false, the program immediately exits and prints an error message indicating the line of source code containing the assertion that failed.

```
// test_sort2.cpp
#include <iostream>
using namespace std;
#include <cassert>
#include "sort.h"
int main()
{
  int i;
  int a[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
  int b[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

  cout << "test selection sort" << endl;
  selection_sort(a, 10);
  for (i=0; i<9; ++i) {
    assert(a[i] <= a[i+1]);
  }
  cout << "selection sort passed" << endl;
  cout << "test merge sort" << endl;
  merge_sort(b, 10);
  for (i=0; i<9; ++i) {
    assert(b[i] <= b[i+1]);
  }
  cout << "merge sort passed" << endl;
  return 0;
}
```

In order to use the C++ `assert` macro, you must include the `cassert` header file. Unlike Python's unit testing framework which indicates that tests pass, using this simple strategy will not produce any output if all the tests pass. If you want output, you can place output statements after each `assert` statement or after a group of `assert` statements to indicate that tests pass as we did in our example. Remember that output is buffered by the operating system and may not appear if the program crashes before the operating system sends the buffered output to the screen. Outputting a new line using `endl` forces the buffer to be flushed, so always use an `endl` at the end of output statements when testing code.

For files that test a number of functions or class methods, you may want to create a separate test function that tests each method and then have the `main` function call each of the test functions. This would be similar to the Python unit testing framework calling all the methods that start with the four characters `test`.

8.15 The Scope and Lifetime of Variables

The *scope* of a variable is the section of source code where it can be accessed and the *lifetime* of a variable is the execution time period starting when the memory for the variable is allocated and ending when it is deallocated. The scope and lifetime of variables in C++ is similar to Python. The scope of a variable in C++ is the block of code in which it is declared. If an inner block declares a variable with the same name as one declared in the outer block, the variable declared in the outer block is not accessible inside that inner block. The following example (that outputs `2 1`) demonstrates this, but for readability reasons, it is not recommended that you declare two different variables with the same name in different blocks of the same function.

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

int main()
{
  int x = 1;

  {
    int x = 2;
    cout << x << " ";
  }
  cout << x << endl;
  return 0;
}
```

Many programmers prefer to only declare variables at the top of a function and the scope of these variables is the function body. As mentioned in section 8.10, you can also declare the loop variable inside the `for` statement and that variable is accessible only inside the body of the loop.

The lifetime of automatic C++ variables starts when the function declaring the variable begins execution and ends when the function completes. Each time a function is called, memory for its automatic variables is allocated on a stack and when the function ends, the memory is deallocated from the stack. This means that the local automatic variables of a function are usually bound to a different memory location each time the function is called and thus, do not remember the value they had the previous time the function was called. If you need a function's local variable to have a "history" and remember its value from the previous call, declare it with the `static` prefix. The following example uses the local static variable `count` to keep track of how many times the function is called. The lifetime of static variables is the lifetime of the program. When the program is started, memory for the variable `count` is allocated and initialized to zero based on the statement inside the function. That same memory location is used for the variable `count` until the program ends; the initialization to zero is executed only once when the memory is first allocated for the variable, not each time the function is called. The scope of the variable `count` is inside the function `f`, but its lifetime is from the start of program execution until the program ends.

```
void f()
{
  static int count = 0;

  count++;
}
```

8.16 Common C++ Mistakes by Python Programmers

Some common mistakes that Python programmers make when learning C++ are

- forgetting the semicolon after each statement
- putting a semicolon at the end of a `for` statement or after the Boolean expression for an `if` or `while` statement
- putting a colon at the end of a `for` statement or after the Boolean expression for an `if` or `while` statement

- forgetting the braces to mark blocks or putting a semicolon after a brace that marks a block of code
- forgetting the parentheses around Boolean expressions in `if`, `while`, and `do while` statements
- forgetting to put the data type in front of each formal parameter in a function/method; for example, writing `void f(int a, b)` instead of the correct `void f(int a, int b)`
- putting the semicolon after the right parenthesis marking the end of a function's parameters when you are writing the code or forgetting that a semicolon is required after the right parenthesis when defining the function prototype but not writing the code
- attempting to directly assign one array variable to another; you must assign each element individually, typically with a loop or nested loops in the case of multi-dimensional arrays

8.17 Additional C++ Topics (Optional)

The topics covered in this section are included to provide a more detailed, but still not complete, introduction to C++ for the interested reader. An understanding of the topics covered in this section is not required to understand the other chapters in this book.

8.17.1 The C++ Switch Statement

C++ supports another decision statement that Python does not have. The C++ `switch` statement is less general than the `if` statement. Any statement written with a `switch` statement can be written using `if` and `else if` statements, but not all `if` statements can be written as `switch` statements. The following C++ code shows an example of a `switch` statement.

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

int main()
{
  int choice;
```

```
  cout << "enter your choice of 1, 2, 3, 4: ";
  cin >> choice;

  switch(choice) {
  case 1:
    cout << "you chose 1\n";
    break;
  case 2:
    cout << "you chose 2\n";
    break;
  case 3:
    cout << "you chose 3\n";
    break;
  case 4:
    cout << "you chose 4\n";
    break;
  default:
    cout << "you made an invalid choice";
  }
  return 0;
}
```

As the example demonstrates, the keyword `switch` is used followed by an expression inside parentheses. The expression must be an *ordinal* value which for our purposes means its type must be `int`, `char`, or `bool`. The expression cannot be a floating point value or a string. The keyword `case` is used to list one of the possible values for the expression. If the value of the expression (`choice` in our example) matches the `case` value then the code under that `case` statement is executed. The execution continues until a `break` statement is encountered or the end of the `switch` statement is reached. When a `break` statement is reached, execution continues with the statement after the ending brace for the `switch` statement (`return 0` in our example). The keyword `default` is used to indicate the code that is to be executed if the expression does not match any of the `case` statements.

Since the `break` statement is required to indicate the end of the code to be executed when a `case` statement matches, you can use this fact to write code such as the following:

```
// switch2.cpp
#include <iostream>
using namespace std;

int main()
{
  int choice;
```

```
  cout << "enter your choice of 1, 2, 3, 4: ";
  cin >> choice;

  switch(choice) {
  case 1:
  case 2:
    cout << "you chose 1 or 2\n";
    break;
  case 3:
  case 4:
    cout << "you chose 3 or 4\n";
    break;
  default:
    cout << "you made an invalid choice";
  }
  return 0;
}
```

Since the `break` statement is required to change the flow of execution, forgetting the `break` statement does not create a syntax error but can be a semantic error as the following example shows:

```
// switch3.cpp
#include <iostream>
using namespace std;

int main()
{
  int choice;
  cout << "enter your choice of 1, 2, 3, 4: ";
  cin >> choice;
  switch(choice) {
  case 1:
    cout << "you chose 1\n";
    break;
  case 2:
    cout << "you chose 2\n";
  case 3:
    cout << "you chose 3\n";
    break;
  case 4:
    cout << "you chose 4\n";
    break;
  default:
    cout << "you made an invalid choice";
  }
  return 0;
}
```

If you enter 2 when running this program it outputs both `you chose 2` and `you chose 3`. You should be able to convert each of these `switch` statements to an `if` statement with the same semantics. As we mentioned earlier, one specific value must follow a `case` statement. You cannot write: `case (choice > 0 && choice < 3):`. The `switch` statement is not commonly used because of these restrictions, although it can be used for menu choices as our examples showed. Another important point to notice is that braces are not used to mark the blocks of code under a `case` statement. This is an inconsistency with how C++ marks blocks of code.

8.17.2 Creating C++ Namespaces

You can create your own namespace using the `namespace` keyword. The following example demonstrates a namespace.

```
namespace searches
{
  // function/class definitions
  void binary_search()
  {
    // code here
  }
}
```

To access the function `binary_search` outside of the namespace block, you have three choices. You can refer to it using the full name `searches::binary_search` each time you want to access it. Another option is to place the statement `using namespace searches` at the top of your file. This allows you to refer to all the functions, classes, etc. defined in the `searches` namespace without prefixing them with `searches::`. This corresponds to the Python statement `from searches import *`. The third option is to put the statement `using searches::binary_search` at the top of your file. This is similar to the Python statement `from searches import binary_search`. This C++ version of the `using` statement allows you to access the `binary_search` function without the need for the `searches` prefix in your code, but any other names defined in the `searches` namespace that you want to access would need to be specified with the `searches::` prefix.

8.17.3 Global Variables

C++ also supports global variables although the use of global variables is generally bad design. One exception to this is that constants are commonly defined as global variables. The lifetime of any global variable is the entire execution time of the

program. To create a global variable, define it at the top of the file outside of any function blocks. A global variable is accessible in any functions in that file and can be accessed in other files if those files declare the variable with an `extern` prefix. If you wish to make a variable accessible only inside the current file, define it with the `static` prefix. In formal terms, the scope of global variables defined with the `static` prefix is the file it is declared within. The memory for global and static variables is allocated when the program is loaded into memory just before its execution starts, and the same memory location is used for global and static variables throughout the entire execution of the program. As you may have noticed, the keyword `static` has multiple meanings depending on the context in which it is used.

Only one file that is part of a program may define a global variable with a specific name (just as there can be only one function with a specific name defined per program), but any number of files may declare that variable `extern` and access the global variable. This is the issue that there can be many declarations, but only one definition. The following example with three files demonstrates a global and a static variable. It also demonstrates another use of `extern` to indicate that the functions `f` and `g` with the specified prototypes exist in another file; this is not recommended and instead you should use header files as discussed in subsection 8.12.1. In either case, you will get an error during the linking phase of building the executable code if the functions cannot be found in one of the compiled object files or the global variables are defined non-`extern` in more than one file.

```
// file1.cpp
int x; // this global variable is potentially accessible
       // in any file linked with file1.o
const double PI=3.141592654; // global constant
static int y; // this variable is only accessible in file1.cpp

extern void f();
extern void g();

int main()
{
  x = 2;
  y = 3;
  f(); // calls f defined in another file
  g(); // calls g defined in another file
  return 0;
}
```

```
// file2.cpp
extern int x; // this allows access to the global variable x defined
              // in another file
void f()
{
  x = 3; // sets x declared in another file (file1.cpp in this case)
}
```

```
// file3.cpp
extern int x;

void g()
{
  x = 4; // sets x declared in another file
}
```

8.18 Chapter Summary

This chapter covers the basic syntax and semantics of much of the C++ language assuming you understand Python. Here is a summary of some of the important concepts.

- C++ code is compiled while Python uses a hybrid technique of compiling to byte code and interpreting the byte code.
- C++ requires you to declare all variable names with a specified type; the built-in types are `int`, `char`, `float`, `double`, and `bool`.
- C++ uses braces, `{}`, to mark blocks of code.
- C++ requires parentheses to be placed around the Boolean expression for the `if`, `while`, and `do while` statements.
- C++ uses the two words `else if` instead of the `elif` keyword that Python uses.
- C++ supports a basic array type for storing groups of data of the same type. C++ arrays are similar to Python lists, but C++ arrays are not a class and thus only support the use of brackets to access individual elements; you cannot slice, concatenate, or assign entire arrays with one statement.
- A declaration indicates the type for an identifier name, while a definition allocates memory (for a variable or the code for a function or method).

- C++ supports two parameter passing methods: pass by value which copies the parameters and pass by reference which causes the formal parameters to refer to the same memory locations as the actual parameters.
- C++ arrays are automatically passed by reference.
- Header files are used to declare functions and global variables; we will learn how header files are used with classes in the next chapter.
- The scope of a variable refers to the section of code in which a variable can be accessed; the lifetime of a variable is the time during the execution of the program in which a specific memory location is associated (bound) to the variable.

8.19 Exercises

True/False Questions

1. All C++ programs must have a function named `main`.
2. Any variable used in a C++ program must be declared with a type before it can be used.
3. A C++ function must return a value.
4. A C++ program that compiles will output the results that you intend it to.
5. A C++ program that does not compile can be executed.
6. If the C++ compiler outputs a warning, it will never compile the program.
7. C++ compiler warnings should be ignored.
8. If you compile a C++ program using the Linux operating system on an Intel chip, you can execute the generated program on a computer running the Windows operating system on the same Intel chip.
9. For simple text-based programs you can usually recompile a C++ program on different architectures and operating systems without changing your code.
10. In general, a compiled C++ program will execute faster than a similar Python program on the same computer.