To be able to determine which variables are global, which variables are local, and which variables are nonlocal.

To be able to differentiate between declarations and definitions.

To be able to differentiate between static and automatic variables.

To be able to define and invoke a numeric value-returning function to implement a specified task.

To be able to define and invoke a Boolean value-returning function to implement a specified task.

To be able to modify the case study program.
Chapter 9: Assignment Cover Sheet

Name __________________________________________ Date _______________________
Section _________________________________________

Fill in the following table showing which exercises have been assigned for each lesson and check what you are to submit: (1) lab sheets, (2) listings of output files, and/or (3) listings of programs. Your instructor or teaching assistant (TA) can use the Completed column for grading purposes.

<table>
<thead>
<tr>
<th>Activities</th>
<th>Assigned: Check or list exercise numbers</th>
<th>Submit (1)</th>
<th>(2)</th>
<th>(3)</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prelab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prelab Assignment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inlab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-1: Check Prelab Exercises</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-2: Static and Automatic Variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-3: Value-Returning and Void Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-4: Test Plan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-5: Debugging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lesson 9-6: Case Study Maintenance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Postlab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prelab Activities

Review

In Chapter 8, we said that local variables are those declared within a block. The block does not have to be the body of a function—local identifiers can be declared within any block. In a large program, there may be several variables with the same identifier—name, for example. How do we know which variable is meant and where each variable is accessible? The answers to these questions are provided by scope rules.

Scope of an Identifier

There are four categories of scope for an identifier in C++: class scope, local scope, namespace scope, and global scope. Class scope is defined in Chapter 11. Local scope is the scope of an identifier defined within a block and extends from the point where the identifier is declared to the end of the block. Global scope is the scope of an identifier declared outside all functions and extends to the end of the file containing the program.

Function names in C++ have global scope. Once a function name has been declared, any subsequent function can call it. In C++ you cannot nest a function definition within another function. When a function defines a local identifier with the same name as a global identifier, the local identifier takes precedence. That is, the local identifier hides the existence of the global identifier. For obvious reasons this principle is called name precedence or name hiding.

The rules of C++ that govern who knows what, where, and when are called scope rules.

1. A function name has global scope.
2. The scope of a global identifier extends from its declaration to the end of the file in which it is defined.
3. The scope of a parameter is the same as the scope of a local variable declared in the outermost block of the function body.
4. The scope of a local identifier includes all statements following the declaration of the identifier to the end of the block in which it is declared and includes any nested blocks unless a local identifier of the same name is declared in a nested block.
5. The scope of an identifier begins with its most recent declaration.

The last three rules mean that if a local identifier in a block is the same as a global or nonlocal identifier to the block, the local identifier blocks access to the other identifier. That is, the block’s reference is to its own local identifier.

To summarize, any variable or constant declared outside all functions is a global identifier. Any global identifier is known (and can be accessed directly) by any function that does not declare a variable or constant with the same name. Local identifiers of a function are nonlocal (but accessible) to any block nested within it.

If global identifiers are accessible to all functions, why don’t we just make all identifiers global and do away with parameter lists? Good programming practice dictates that communication between the modules of our program should be explicitly stated. This practice limits the possibility of one module accidentally interfering with another. In other words, each function is given only what it needs to know. This is how we write “good” programs.
The avoidance of global variables is especially critical in a team environment, where one person should not have to check with everyone else before using or changing a variable because it might be global.

Namespaces
As a concept, namespace is the same as scope. In C++, namespace is a language feature that allows a programmer to create a named scope. The authors of <iostream>, for example, enclosed the definitions and declarations within a namespace called std. To access streams cin and cout and the manipulator endl, we use the statement

using namespace std;

near the front of our programs. This statement tells the compiler that all identifiers defined in namespace std (defined in file iostream) are accessible to the parts of the program that are within the scope of the using directive.

Declarations and Definitions
In Chapter 8, we used a function prototype to tell the compiler that a function with certain parameters would be defined later. The function prototype is a declaration that is not also a definition. The function definition includes a heading and a body. One need for a function prototype comes about because most C++ programmers put function main physically first in the file. main invokes other functions, so the names of the other functions and their parameter types must be known before function main is compiled. In our example, if we had put the function definition physically before main, the prototype would not have been necessary, and the function definition would have been both a declaration and a definition. However, function prototypes cannot be eliminated simply by moving main: After all, we might have a function that calls a second function, with that second function also calling the first function.

The distinction between a declaration and a definition is that memory space is reserved in a definition. In a function definition, the space for the compiled code of the function body is reserved. There are times when we want to declare a variable but not define it. That is, we want to let the compiler know about a global variable defined in another file. Look at the following two statements:

int myValue;
extern int anotherValue;

The first statement is both a declaration and a definition. Space is reserved for the variable myValue of data type int. The second statement is a declaration only. Your program can reference anotherValue, but the compiler does not reserve space for it. The reserved word extern tells the compiler that anotherValue is a global variable located in another file.

More about Variables
The lifetime of a variable is the time during program execution when the variable has storage assigned to it. The lifetime of a global variable is the entire execution of the program. The lifetime of a local variable is the execution of the block in which it is
declared. Sometimes we want to have the value of a local variable remain the same between calls to the same function. For example, to know how many times a function is called during the execution of a program, we would like to use a local function variable as a counter and increment it each time the function is called. However, if space is allocated each time the function is called and deallocated when the function finishes executing, we can’t guarantee that the space for the local variable will be the same for each function invocation. In fact, it most likely would not be.

To get around this problem, C++ lets the user determine the lifetime of each local variable by assigning it a storage class: static or automatic. If you want the lifetime of a local variable to extend to the entire run of the program, you preface the data type identifier with the reserved word `static` when the variable is defined. The default storage class is automatic, in which storage is allocated on entry and deallocated on exit from a block.

Any variable may be initialized when it is defined. This is the fourth way to assign a value to a place in memory. (The first three were discussed in Chapter 4.) To initialize a variable, follow the variable identifier with an equal sign and an expression. The expression is called an *initializer*. The initializing process differs depending on whether the variable being initialized is static or automatic. For a static variable, the initializer must be a constant expression, and its value is stored only once. For an automatic variable, the initializer can be any expression, and its value is stored each time the variable is assigned storage.

### User-Defined Value-Returning Functions

With void functions, we use the function name as a statement in our program to represent the action that the function performs. Value-returning functions are used when the action returns a single value and that value is used in an expression. For example, let’s write a function that returns the smallest of its three input values.

```cpp
// Program PrintMin prints the smallest of three input values.
#include <iostream>
using namespace std;

int Minimum(int, int, int); // Function prototype
// Returns the minimum of three distinct values.

int main()
{
    int one;
    int two;
    int three;

    cout << "Input three integer values; press return." << endl;
    cin >> one >> two >> three;
    cout << "The minimum value of " << one << ", " << two
         << ", and " << three << " is "
         << Minimum(one, two, three) << "." << endl;
    return 0;
}

//****************************************************************************
```
int Minimum(int first, int second, int third)
// Pre:  Input values are distinct; there are no duplicates.
// Post: Returns minimum of three distinct int values.
{
    if (first < second && first < third)
        return first;
    else if (second < first && second < third)
        return second;
    else if (third < first && third < second)
        return third;
}

Output from three runs:

Input three integer values; press return.
2 3 4
The minimum value of 2, 3, and 4 is 2.

Input three integer values; press return.
4 3 2
The minimum value of 4, 3, and 2 is 2.

Input three integer values; press return.
-2 -3 6
The minimum value of -2, -3, and 6 is -3.

The function prototype declares a function of data type int. This means that the type of the value returned to the calling function is int. Because a value-returning function always sends back one value, we designate the type of the value before the function name. We call this data type the function type. However, function value type, function return type, or function result type is more accurate.

In this example, the function invocation occurs in the output statement in function main. Function Minimum is invoked, and the value is returned and immediately sent to the output stream. There are three parameters: first, second, and third. The three arguments are one, two, and three. Notice that the documentation on the function prototype is different from the documentation on the function definition.
Chapter 9: Prelab Assignment

Read program Scope carefully and then complete Exercises 1 through 6.

// Program Scope demonstrates scope rules and lifetime classes.
#include <iostream>
#include <fstream>
using namespace std;

int counter;
int sum = 0;
int number;
ifstream inNums;

void SumNumbers(ifstream& inFile, int& answer);
// Sums the numbers on inFile.

int main ()
{
    inNums.open("numeric.dat");
    int sum = 0;
    SumNumbers(inNums, sum);
    cout << "Output from first call to SumNumbers" << endl;
    cout << "Sum is " << sum << endl;
}

SumNumbers(inNums, sum);
cout << "Output from second call to SumNumbers" << endl;
cout << "Sum is " << sum << endl;
return 0;

void SumNumbers(ifstream& inFile, int& answer)
{
    static int counter = 1;
    while (counter <= 4)
    {
        inFile >> number;
        answer = answer + number;
        counter++;
    }
}
Exercise 1: File numeric.dat contains the following values: 10 20 30 40 10 20 30 40. What is printed?

Exercise 2: List the global identifiers.

Exercise 3: Circle each of the blocks in program Scope and number them from top to bottom.

Exercise 4: List the local identifiers and state the block(s) in which each is accessible.

Exercise 5: List the automatic local variables.

Exercise 6: List the static local variables.
Lesson 9-1: Check Prelab Exercises

Exercise 1: Run program `Scope` to check your answers. Were your answers correct? If not, explain what you did wrong.

Exercise 2: `counter`, `sum`, `number`, `inNums`, and `SumNumbers`.

Exercise 3:
```c++
int main ()
{
    
    
    void SumNumbers(ifstream & inPile, int & answer)
    {
        
    }
}
Exercise 4: Block 1 has no local identifiers. Block 2 has local identifier `sum`, which hides global identifier `sum`. Block 3 has three local identifiers: `inFile`, `answer`, and `counter`.

Exercise 5: Automatic local variable: `sum` (local to 2). All of the parameters are also automatic variables.

Exercise 6: Static local variable: `counter` (local to 3). All global variables are static.
Lesson 9–2: Static and Automatic Variables

Use program Differences for Exercises 1 and 2.

// Program Differences is for investigating the differences
// between automatic and static variables.

#include <iostream>
using namespace std;

void TestLocals();

int main ()
{
    TestLocals();
    TestLocals();
    TestLocals();
    return 0;
}

// ***************************************

void TestLocals()
{
    /* TO BE FILLED IN */
}

Exercise 1: Function TestLocals defines a local variable, count, and initializes it to 1. The contents of count are incremented and sent to the output stream. Write this code and run program Differences. Record your output. Was it what you expected?

Exercise 2: Change function TestLocals so that local variable count is a static variable. Rerun your program and show your output. Was it what you expected?

Exercise 3: Explain the difference between automatic and static variables.
Lesson 9–3: Value-Returning and Void Functions

Exercise 1: Lesson 6–4, Exercise 1, contains a loop nested within a loop. The outer loop prompts for and inputs a digit. The inner loop sums the integers from zero through the digit. Rewrite your solution to this task so that the summing of the digits from zero through the input digit is done in an `int` function, `SumDigits`. The digit should be an input parameter. Rerun your changed version with the same data. The results should be the same.

Exercise 2: Write a program with an `int` function, `Minimum`, that finds the minimum value on a file. Use file `numbers.dat` as input.

Minimum value is ________________.

Exercise 3: In Exercise 2, did you access the file name globally or did you pass the file name as a parameter to function `Minimum`? If you passed the file name, you had a reference parameter to a value-returning function. Both accessing a variable globally and having a reference parameter to a value-returning function are bad style. Should this function have been a void function with two reference parameters? Justify your answer.
Exercise 4: Program Convert is an old friend. This version uses an int function to convert temperatures from Fahrenheit to Celsius or from Celsius to Fahrenheit. Examine program Convert carefully. The next exercises ask you to modify it.

```
// Program Convert converts a temperature from Fahrenheit to
// Celsius or a temperature from Celsius to Fahrenheit
// depending on whether the user enters an F or a C.

#include <iostream>
using namespace std;

int ConvertedTemp(int tempIn, char letter);
// If letter is a 'C,' tempIn is converted from Celsius
// to Fahrenheit; otherwise, tempIn is converted from
// Fahrenheit to Celsius.

int main ()
{
    char letter; // Place to store input letter
    int tempIn; // Temperature to be converted
    cout << "Input Menu" << endl << endl;
    cout << "F:  Convert from Fahrenheit to Celsius" << endl;
    cout << "C:  Convert from Celsius to Fahrenheit" << endl;
    cout << "Q:  Quit" << endl;
    cout << "Type a C, F, or Q; then press return." << endl;
    cin  >> letter;
    while (letter != 'Q')
    {
        cout << " Type an integer number, and press return." << endl;
        cin >> tempIn;
        if (letter == 'F')
            cout << "Fahrenheit to Celsius" << endl;
        else
            cout << "Celsius to Fahrenheit" << endl;
        cout << "Temperature to convert: " << tempIn << endl;
        cout << "Converted temperature: " << ConvertedTemp(tempIn, letter) << endl;
        cout << "Type a C, F, or Q; then press return." << endl;
        cin  >> letter;
    }
    return 0;
}

// *****************************************************

int ConvertedTemp(int tempIn, char letter)
{
    if (letter == 'C')
        return (9 * tempIn / 5) + 32;
    else
        return 5 * (tempIn - 32) / 9;
}
Run program Convert using the data shown in the table below.

<table>
<thead>
<tr>
<th>Code</th>
<th>Temperature</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>________</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
<td>________</td>
</tr>
<tr>
<td>c (lowercase)</td>
<td>100</td>
<td>________</td>
</tr>
<tr>
<td>C</td>
<td>57</td>
<td>________</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>________</td>
</tr>
<tr>
<td>F</td>
<td>42</td>
<td>________</td>
</tr>
<tr>
<td>F</td>
<td>212</td>
<td>________</td>
</tr>
<tr>
<td>F</td>
<td>134</td>
<td>________</td>
</tr>
<tr>
<td>Q</td>
<td>Run terminated</td>
<td></td>
</tr>
</tbody>
</table>

Exercise 5: Enclose the menu in a void function with no parameters. Rerun your program to check that the results are the same.

Exercise 6: Did you notice that the wrong answer was returned when a lowercase c was input? Program Convert has no error checking. Incorrect input just produces an incorrect answer. Write a Boolean function, Error, that takes letter and returns true if letter is not an uppercase C, F, or Q and false otherwise. Use Error to check for bad data immediately after letter is read. If letter is not correct, write an error message and prompt again for letter. Run the original program and your revised version using lowercase c and uppercase X as inputs.

Show output from the original version.

<table>
<thead>
<tr>
<th>Code</th>
<th>Temperature</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>c (lowercase)</td>
<td>0</td>
<td>________</td>
</tr>
<tr>
<td>X</td>
<td>100</td>
<td>________</td>
</tr>
</tbody>
</table>

Show output from the revised version.

<table>
<thead>
<tr>
<th>Code</th>
<th>Temperature</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>c (lowercase)</td>
<td>0</td>
<td>________</td>
</tr>
<tr>
<td>X</td>
<td>100</td>
<td>________</td>
</tr>
</tbody>
</table>
Exercise 7: Restricting the input to uppercase letters in program Convert is unrealistic. Revise the program so that it accepts both uppercase and lowercase letters as input. (Lesson 9-4, Exercise 1, asks you to write a test plan for the revised version of program Convert.)
Lesson 9-4: Test Plan

Exercise 1: Design a test plan for program \texttt{Convert} as revised in Lesson 9-3, Exercise 7.

<table>
<thead>
<tr>
<th>Reason for Test Case</th>
<th>Input Values</th>
<th>Expected Output</th>
<th>Observed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exercise 2: Implement the test plan designed in Exercise 1. You may show the results in the chart in Exercise 1.
Lesson 9–5: Debugging

Exercise 1: Function Swap takes two values as input and returns them in reverse order. This little utility routine is very useful, but it doesn’t seem to work correctly. Can you fix it? Program Driver is written to test function Swap. A program whose only purpose is to test a subprogram is called a driver. The driver reads in two values, calls Swap with the two input values, and writes out the result. Run Driver using 10 and 15 as the input values. What is printed?

Exercise 2: This answer looks correct but the program has a bug in it—take our word for it. Search until you find an error, correct it, and run the program again. What is printed?

Exercise 3: We bet there is still a problem with your program. Keep trying. What error did you find this time?
Lesson 9–6: Case Study Maintenance

Name __________________________________________ Date _______________________
Section _________________________________________

Exercise 1: Compile and run program Profile (profile.cpp), using data of your choice.

Exercise 2: The tables used in program Health were not consistent in form because they came from different sources; that is, some were written in a form that could be directly coded in C++ and some were not. For example the cholesterol charts used “and.” In coding them, they all had to be changed into a cascading if form. Here are the tables themselves in that form.

<table>
<thead>
<tr>
<th>BMI</th>
<th>Interpretation</th>
<th>High Risk Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20</td>
<td>Underweight</td>
<td></td>
</tr>
<tr>
<td>≤ 25</td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>≤ 30</td>
<td>Overweight</td>
<td>XXXX</td>
</tr>
<tr>
<td>&gt; 30</td>
<td>Obese</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HDL</th>
<th>Interpretation</th>
<th>High Risk Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 40</td>
<td>Too low</td>
<td>XXXX</td>
</tr>
<tr>
<td>&lt; 60</td>
<td>Is okay</td>
<td>XXXX</td>
</tr>
<tr>
<td>≥ 60</td>
<td>Excellent</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LDL</th>
<th>Interpretation</th>
<th>High Risk Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 100</td>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>&lt; 130</td>
<td>Near optimal</td>
<td></td>
</tr>
<tr>
<td>&lt; 160</td>
<td>Borderline high</td>
<td></td>
</tr>
<tr>
<td>&lt; 190</td>
<td>High</td>
<td>XXXX</td>
</tr>
<tr>
<td>≥ 190</td>
<td>Very high</td>
<td>XXXX</td>
</tr>
</tbody>
</table>
Systolic Interpretation

<table>
<thead>
<tr>
<th>Systolic</th>
<th>Interpretation</th>
<th>High Risk Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 120</td>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>&lt; 130</td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>&lt; 140</td>
<td>Normal high</td>
<td></td>
</tr>
<tr>
<td>&lt; 160</td>
<td>Stage 1 hypertension</td>
<td>XXXX</td>
</tr>
<tr>
<td>&lt; 180</td>
<td>Stage 2 hypertension</td>
<td>XXXX</td>
</tr>
<tr>
<td>≥ 180</td>
<td>Stage 3 hypertension</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

Diastolic Interpretation

<table>
<thead>
<tr>
<th>Diastolic</th>
<th>Interpretation</th>
<th>High Risk Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 80</td>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>&lt; 85</td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>&lt; 90</td>
<td>High normal</td>
<td></td>
</tr>
<tr>
<td>&lt; 100</td>
<td>Stage 1 hypertension</td>
<td>XXXX</td>
</tr>
<tr>
<td>&lt; 110</td>
<td>Stage 2 hypertension</td>
<td>XXXX</td>
</tr>
<tr>
<td>≥ 110</td>
<td>Stage 3 hypertension</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

Five items are checked for each person. Enhance the program to keep track of how many of a person’s values fall into a high-risk (warning) category. The high-risk categories are those marked with X’s in the third column. Examples are shown in the following table.

<table>
<thead>
<tr>
<th>Patient</th>
<th>BMI</th>
<th>HDL</th>
<th>LDL</th>
<th>Systolic</th>
<th>Diastolic</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>25</td>
<td>50</td>
<td>130</td>
<td>140</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>#2</td>
<td>31</td>
<td>20</td>
<td>190</td>
<td>160</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>#3</td>
<td>27</td>
<td>65</td>
<td>170</td>
<td>139</td>
<td>95</td>
<td>2</td>
</tr>
</tbody>
</table>

For each person, print the number of high-risk categories into which that individual falls. Compile and run the program with sufficient data to check that each combination of high-risk categories is represented.
Exercise 3: Design a summary table like the one shown in Exercise 2. Code, compile, and run the program using the data designed for Exercise 2.

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Postlab Activities

Exercise 1: Along with eating a heart-healthy diet, everyone should get regular exercise. You are attending an aqua jogging class. After you have been jogging for 15 minutes, the instructor stops the class and asks everyone to take his or her pulse for six seconds. (She times you.) She then asks if you are in your target zone. To help the class out, you decide to program a target-zone calculator. Each person's target zone is between 60 percent and 75 percent of his or her predicted maximal heart rate in beats per minute. A good approximation to this maximal heart rate is 220 minus the person's age.

Write a function, InTheZone, that takes a person's age and the six-second pulse rate and returns true if the person is within his or her zone and false otherwise. Keep prompting for and reading an age and a pulse rate until the user enters a negative value. You must use at least one value-returning function in this assignment.

Exercise 2: Write a driver and a test plan for function InTheZone. Implement your test plan.

Exercise 3: In Postlab Exercise 6-6, you were asked to create a table comparing Celsius temperatures with the actual Fahrenheit equivalent and an approximated equivalent. Redesign that program and use float functions to calculate the actual value and the approximated value. Run your program with the same data.

Exercise 4: Add a column to the table in Exercise 3 that shows the difference between the actual Fahrenheit equivalent and the approximation. This change makes the use of value-returning functions as specified in Exercise 3 inappropriate. Rewrite the float functions as void functions and explain why value-returning functions are no longer appropriate.