PROC FCMP
User-Defined Functions:

An Introduction to the SAS® Function Compiler

Troy Martin Hughes

The correct bibliographic citation for this manual is as follows: Hughes, Troy Martin. 2024. *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler*. Cary, NC: SAS Institute Inc.

PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler

ISBN 978-1-68580-006-2 (Paperback)
ISBN 978-1-68580-007-9 (Web PDF)
ISBN 978-1-68580-028-4 (EPUB)
ISBN 978-1-68580-008-6 (Kindle)

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

April 2024

Dedication page photo credit: Zeke Torres

Contents

About This Book

Preface

Software development represents a tremendous investment of resources; business needs must be identified and discussed, and code must be designed, written, tested, documented, deployed for use, and ultimately maintained. To maximize return on investment, software should be reused as many times as possible, by as many users as possible, for as long as possible—or at least while it continues to deliver business value.

To this end, wrapping software functionality inside modular functions is a rewarding best practice that encourages software reuse. This *software modularity* facilitates software configuration, in which varied inputs (arguments) produce dynamic output (return values). Configurable modules replace unnecessary hardcoding, and facilitate repeatable, reusable software components that can meet the needs of diverse users and diverse use cases.

The SAS language includes hundreds of built-in functions—from ABS, which calculates the absolute value of a number, to ZIPSTATE, which converts a ZIPCODE into its corresponding state abbreviation. But every programming language has its limits, and where no built-in function exists to provide some needed functionality, a user-defined function can be built to deliver that functionality and effectively *extend* the programming language.

This text introduces PROC FCMP—the SAS Function Compiler—the procedure with which SAS practitioners can create user-defined functions and subroutines. These modular, callable software components complement the diverse array of SAS built-in functions and provide a richer, more expansive development environment in which to build SAS software.

User-defined functions improve the quality of SAS software by extracting complex logic, business rules, and other operations from DATA steps. Encapsulating this functionality inside functions, yields more maintainable, readable, reusable software. User-defined functions also improve the quality of the development environment itself. The productivity of SAS practitioners surges because we are able to reuse user-defined functions rather than having to reinvent the wheel.

To those plucky practitioners, intent on advancing your SAS repertoire and resume, this book is for you! It introduces the FCMP procedure, including its use cases, syntax, best practices, and benefits. Hardcoding puts the “SAS” in disaster, but it can be averted through flexible, reusable user-defined functions!

What Does This Book Cover?

You will be introduced to the FCMP procedure and instructed how to build user-defined functions—callable, reusable, beautifully bite-sized chunks of software functionality that fundamentally change how you conceptualize, design, and develop SAS software.

But first, you will be introduced to functions themselves so that you can see how functions improve the quality of not only software but also the software development environment. And with this foundation, FCMP syntax is incrementally demonstrated through requirements-based examples. You will walk away having gained the ability to examine your own software business needs and evaluate whether, where, and how you can implement user-defined functions to overcome obstacles, provide analytic insight, and deliver business value.

Organization

PROC FCMP User-Defined Functions is intended to be read cover to cover, as concepts, syntax, and examples build incrementally from one chapter to the next. For those interested in learning about a specific FCMP statement, function, subroutine, or other syntactical element, a comprehensive index facilitates direct access to the material.

Chapter 1 introduces functions in a programming-language-agnostic sense, including both built-in and user-defined functions. SAS functions and subroutines are introduced and contrasted with SAS procedures. Function nomenclature is defined in this chapter, including software quality characteristics, which are referenced throughout the remainder of the text.

Chapter 2 introduces basic FCMP syntax, including how to build simple functions and subroutines. The majority of the chapter focuses on function communication, including how to transfer data *to* a function, and how to retrieve results *from* a function. Differences between DATA step syntax and the FCMP procedure are also explored.

Chapters 3 and 4 introduce the SAS array and hash object, respectively, which are the primary built-in data structures leveraged by user-defined functions. Later chapters rely on these data structures to deliver dynamic functionality while minimizing code complexity and maximizing efficiency.

Chapter 5 introduces the RUN_MACRO and RUN_SASFILE built-in functions, which operate only inside the FCMP procedure. They enable user-defined functions to call SAS macros or to execute SAS programs during a function call. Chapter 6 delves further into RUN_MACRO by demonstrating how it can support data lookup operations.

Chapter 7 focuses on function design, including recursion and memoization. *Recursion* describes the act of a function or subroutine calling itself, and *memoization* describes the retention of results from costly (that is, resource-intensive) function calls to improve software runtime and efficiency.

Chapter 8 demonstrates the interaction between the FCMP procedure and the Python open-source language. The Python Component Object is introduced, which facilitates interoperability by enabling SAS user-defined functions to call Python functions.

Chapter 9 introduces two powerful methods to call user-defined functions. The FORMAT procedure OTHER option is demonstrated, which enables you to design user-defined formats and informats that call user-defined functions. The REPORT procedure COMPUTE block is demonstrated, in which user-defined functions can be called to modify and add dynamic functionality to SAS reports.

Is This Book for You?

This text is intended for intermediate to advanced SAS users who have a firm grasp of the DATA step and who are looking to maximize the potential of their software. Nevertheless, because the majority of DATA step syntax can be run inside the FCMP procedure, FCMP user-defined functions can and should be incorporated early in your SAS career. For this reason, this text gradually introduces the principal built-in data structures of the FCMP procedure—the SAS array and the hash object—to ensure users of all levels can understand and confidently interact with them. Knowledge of the SAS macro language is not a prerequisite to learning the FCMP procedure; however, some examples in this text do incorporate SAS macro statements, functions, variables, and other syntax.

What Should You Know about the Examples?

This book includes tutorials for you to follow to gain hands-on experience with SAS.

Software Used to Develop the Book's Content

All examples in this text require only Base SAS; no other SAS modules are required.

Example Code and Data

You can access the example code and data for this book by linking to its author page at https://support.sas.com/en/books/authors/troy-hughes.html.

SAS OnDemand for Academics

If you are using SAS OnDemand for Academics to access data and run your programs, then please check the SAS OnDemand for Academics page to ensure that the software contains the

product or products that you need to run the code: https://www.sas.com/en_us/software/on-demand-for-academics.html.

We Want to Hear from You

SAS Press books are written *by* SAS Users *for* SAS Users. We welcome your participation in their development and your feedback on SAS Press books that you are using. Please visit sas.com/books to do the following:

- Sign up to review a book
- Recommend a topic
- Request information on how to become a SAS Press author
- Provide feedback on a book

Learn more about this author by visiting his author page at https://support.sas.com/en/books/authors/troy-hughes.html. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

About The Author

Troy Martin Hughes has been a SAS practitioner for more than 20 years; has managed SAS projects in support of federal, state, and local government initiatives; and is a SAS Certified Advanced Programmer, SAS Certified Base Programmer, SAS Certified Clinical Trials Programmer, and SAS Certified Professional V8. He has an MBA in information systems management and additional credentials, including: PMP, PMI-ACP, PMI-PBA, PMI-RMP, SSCP, CSSLP, CISSP, CRISC, CISM, CISA, CGEIT, Network+, Security+, CySA+, CASP+, Cloud+, CSM, CSP-SM, CSD, A-CSD, CSP-D, CSPO, CSP-PO, CSP, SAFe Government Practitioner, and ITIL Foundation. He has given more than 150 presentations, trainings, and hands-on workshops at SAS user group conferences, including SAS Global Forum, SAS Analytics Experience, SAS Explore, WUSS, MWSUG, SCSUG, SESUG, PharmaSUG, BASAS, and BASUG. He is the author of two groundbreaking books that model SAS best practices, including *SAS® Data Analytic Development: Dimensions of Software Quality* (2016), and *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition* (2022). Troy is a U.S. Navy veteran with two tours of duty in Afghanistan.

Learn more about this author by visiting his author page at https://support.sas.com/en/books/authors/troy-hughes.html. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Chapter 1: Introducing Functions

Functions deliver functionality—this much is clear. But what makes a function a function? How do functions differ from other code and software components? And most importantly, why should SAS practitioners learn to build our own (that is, *user-defined*) functions? These and other questions are explored and answered in the following chapters as functions are introduced, including their purpose, value, syntax, construction, and implementation. You will learn how to build functions using the SAS Function Compiler procedure (PROC FCMP), and how to integrate user-defined functions into SAS programs to improve software quality.

Functions are the simple syrup of software, and for those who have never bartended, allow me to explain. Simple syrup is simple—one part water, one part granulated sugar. Mix, heat, stir, dissolve, chill, and incorporate into various cocktails over several hours or days or until the carafe runs dry. Yes, the recipe is straightforward, but you wouldn't want to be caught empty-handed during a hectic happy hour—and making separate syrupy batches for each customer's drink would waste precious time! Of course, the solution is to make the syrup once, test its quality, and reuse it thereafter for effortless rounds of mojitos and daiquiris, improving the efficiency and productivity of any bartender or mixologist.

Just as various cocktails can be concocted by leveraging simple syrup, software, too, is commonly developed by combining components—including reliable, reusable functions that deliver consistent functionality each time they are used. This functionality can be predictably varied or configured through *arguments*—user-supplied input values. In this manner, functions improve software quality by promoting software configurability, reusability, and maintainability. And as the ease with which software can be developed, tested, documented, and maintained increases, developer productivity commensurately increases. Thus, functions operationalize the "working smarter not harder" mindset and improve the quality of not only software itself but also the software development *environment*—the experience of SAS practitioners writing SAS software.

This chapter introduces functions and function-related nomenclature relied upon throughout the text. Two types of callable software modules—functions and subroutines—are compared, contrasted, and disambiguated. SAS built-in functions available in Base SAS are contrasted with user-defined functions. Most importantly, specific characteristics of software quality—namely,

configurability, reusability, and maintainability—are explored, including the role functions play to increase these characteristics. Thus, whereas later chapters introduce the FCMP procedure and its syntax, this chapter makes the business case for designing and implementing user-defined functions that extend the SAS language.

What Is a Function Anyway?

Some discussion of nomenclature should preface any introduction to functions to define and differentiate terminology relied upon throughout later chapters. The International Organization for Standardization (ISO) defines a *function* as a "software module that performs a specific action, is invoked by the appearance of its name in an expression, receives input values, and returns a single value" (International Organization for Standarization 2017). SAS documentation similarly defines a *function* as "a component of the SAS programming language that can accept arguments, perform a computation or other operation, and return a value" (SAS Institute Inc. 2020). In the following subsections, these definitions are further decomposed, explored, and expanded to introduce functions within the SAS language.

Every function is *callable*—that is, built as an independent software module, and *called* (executed) when its name is referenced within code. The *calling module* (or *calling program*) calls a function (the *called module*), and temporarily transfers program control to the function, after which control is returned to the calling module when the function terminates. In SAS, the DATA step typically acts as the calling module (although numerous other methods are demonstrated in this chapter), and the called module always represents a user-defined function or subroutine built and compiled using the FCMP procedure. Calling, callable, and called modules are described subsequently in more detail.

Because so many FCMP syntax elements are identical between functions and subroutines—and benefits are comparable between functions and subroutines—within this text, *function* is used generically to reference both functions and subroutines. *Subroutine* is used only in those rare instances where syntax or functionality differs. In other words, this chapter could be titled "Introducing Functions and Subroutines." When a paragraph decries how "user-defined functions increase the quality of SAS software," you should interpret this as "user-defined functions *and subroutines* increase the quality of SAS software." And they really do!

Functions Versus Functionality

Functions deliver software functionality—they perform some action to effect some result. But software often can be constructed without functions, and nevertheless provide equivalent functionality. Thus, functions differ not so much in *what* they do but in *how* they are structured. As callable software modules, functions are discrete software components (that is, bite-sized

chunks of code) that can be reused over time, and typically configured through parameters to provide flexible results.

Consider the not-too-distant past when Base SAS included the UPCASE function (that converts text to uppercase) but did not have a corresponding LOWCASE function. Frank DiIorio, in his seminal book, notes in a discussion about UPCASE that "There is no analogous function [to UPCASE] to convert to lowercase" (DiIorio 1997). Fortunately, SAS did introduce the LOWCASE function. However, in a pre-LOWCASE world, SAS practitioners would have had to develop customized code to transform text to lowercase.

For example, Program 1.1 converts the Phrase variable to lowercase without calling the LOWCASE function. Instead, the DO loop uses the LENGTH function to assess the length of Phrase and iterates over each character in Phrase. The CHAR function isolates one character at a time, and RANK evaluates the ASCII numeric value of the character. The IF statement evaluates whether a character falls between the ASCII values of 65 and 90 (corresponding to uppercase A through Z in a Windows environment). If so, 32 is added to the ASCII value, and the BYTE function transforms the ASCII value back into its (lowercase) alphabetic equivalent. Finally, the SUBSTR function used on the left-hand side of the equal sign incrementally replaces each uppercase character with its lowercase equivalent.

Program 1.1: Lowercase Functionality in a Non-LOWCASE World

```
data lower;
   length phrase $100;
   phrase = 'SAS Applications Programming: A Gentle Introduction';
   do i = 1 to length(phrase);
      if 65 <= rank(char(phrase,i)) <= 90 then substr(phrase,i,1)
         = byte(rank(char(phrase,i)) + 32);
      end;
   put phrase;
run;
```

The DATA step converts the title of Frank's inimitable book to lowercase, as shown in the SAS log:

```
sas applications programming: a gentle introduction
NOTE: The data set WORK.LOWER has 1 observations and 3 variables.
```

Program 1.1 provides lowercase *functionality* but is not a *function*, as the functionality is not callable, but rather is constructed inside the DATA step. And because this functionality is not callable, the code must be re-created whenever a different variable needs to be converted to lowercase. This becomes a tedious process of copying the DO loop and lowercase functionality whenever a variable needs to be transformed; this repetition is inefficient, and risks the unnecessary introduction of errors.

Fortunately, the LOWCASE built-in function *does* exist, and Program 1.2 produces identical output with far less effort. It is in this manner that one talks about *extending* a programming language

through the addition of functions—because each new function that is defined, whether built-in or user-defined, represents functionality that can be readily called rather than painstakingly re-created in subsequent programs.

Program 1.2: Functionally Equivalent Use of LOWCASE to Transform Text to Lowercase

```
data lower;
   length phrase $100;
   phrase = 'SAS Applications Programming: A Gentle Introduction';
   phrase = lowcase(phrase);
   put phrase;
run;
```

Programs 1.1 and 1.2 are said to be *functionally equivalent*—that is, their results or output are identical; however, they operate using vastly different approaches. Program 1.1 delivers functionality through a DO loop and hardcoded logic, whereas Program 1.2 relies on the LOWCASE function. Program 1.2 is more appealing and inarguably demonstrates better software design because the complexity of the LOWCASE functionality is *abstracted*—hidden from view, and concealed within unseen, proprietary SAS code.

The beauty of abstraction is that it allows the user to focus on the *functionality* that a function delivers rather than the *methods* through which that functionality is delivered. As a SAS practitioner, I do not need to understand the inner workings of LOWCASE, such as whether a DO loop is used or how the case transformation occurs. Moreover, these methods would clutter my DATA step, as demonstrated in Program 1.1, making it more difficult to understand the high-level intent and flow of the program. Thus, Program 1.2 can be said to be more *readable* than Program 1.1, which improves software quality.

The FCMP procedure empowers SAS practitioners to create our own *user-defined* functions. Although FCMP syntax is not discussed yet, Program 1.3 demonstrates the ease with which the logic from the Program 1.1 DATA step can be dropped into the FCMP procedure to create a user-defined function that converts text to lowercase. The Phrase variable has been renamed Str to improve readability, and the remainder of the DO loop is unchanged.

Program 1.3: Functionally Equivalent User-Defined TINY Function

```
* converts character variable to lowercase;
* requires single character parameter <= 100 characters;
* no exception handling for arguments that exceed 100 characters;
* tested and intended for use ONLY in a Windows environment;
proc fcmp outlib=work.funcs.char;
   function tiny(str $) $100;
      do i = 1 to length(str);
         if 65 <= rank(char(str,i)) <= 90 then substr(str,i,1)
            = byte(rank(char(str,i)) + 32);
         end;
      return(str);
      endfunc;
quit;
```

Program 1.3 defines and compiles the TINY user-defined function, whose functionality is approximately equivalent to the LOWCASE built-in function. TINY can be called in the identical fashion as LOWCASE, and Program 1.4 calls TINY and produces results identical to Programs 1.1 and 1.2. Note that the CMPLIB option (described later in greater detail) must be set, which tells SAS where to find user-defined functions.

Program 1.4: Functionally Equivalent Use of TINY to Transform Text to Lowercase

```
options cmplib = work.funcs;

data lower;
   length phrase phrase1 phrase2 $100;
   phrase = 'SAS Applications Programming: A Gentle Introduction';
   phrase1 = lowcase(phrase);
   phrase2 = tiny(phrase);
   put phrase1=;
   put phrase2=;
run;
```

The log demonstrates that LOWCASE and TINY produce identical results:

```
phrase1=sas applications programming: a gentle introduction
phrase2=sas applications programming: a gentle introduction
```

So, why the careful distinction between *identical* results yet *equivalent* functionality? Because SAS user-defined functions, as necessary as they are to building reusable functionality, inherently deliver different (and typically diminished) *performance* than their built-in function counterparts. For example, in designing TINY, no attempt was made to measure or optimize TINY's runtime or utilization of system resources. SAS user-defined functions like TINY are written in Base SAS—a fourth-generation language (4GL) that understandably lacks some of the memory and resource management capabilities that lower-level languages like C, C++, or Java provide. To be clear, this is not a deficiency in the SAS language but rather the result of the SAS application managing lower-level processes. SAS practitioners can focus instead on loftier and, arguably, more interesting pursuits such as data analysis, the production of data products, and data-driven decision-making.

Also note that TINY is said to be "approximately" equivalent to LOWCASE. This caveat acknowledges that although both functions produce identical results given *this* specific input, variability in the data or environment would cause the functions to produce different results. In other words, TINY is less robust and less reliable than LOWCASE. For example, TINY declares a return value having a length of 100, so any character variable passed to TINY that exceeds this threshold will be truncated. This could be described as a failure of *scalability*, one characteristic of software quality, because TINY as currently defined is unable to accommodate longer character values. LOWCASE, on the other hand, is scalable and overcomes these limitations.

TINY also relies on "standard" ASCII character encoding in which the uppercase letters A through Z correspond to the ASCII values 65 through 90—but this encoding is not standard across all

operating environments. For example, TINY would fail on mainframe SAS running on the z/OS platform, which relies on EBCDIC encoding. This inability to provide equivalent functionality across platforms demonstrates a lack of *interoperability*, another characteristic of software quality. This is not to say that user-defined functions inherently lack quality, but rather that potential issues should be identified, and their risks evaluated to determine whether those risks should be mitigated by expanding functionality or improving performance.

For example, TINY could be modified to return longer character values, or to detect the operating system programmatically—but the decision to refactor a function should be made based on the business value those modifications produce or the risks that they mitigate. Thus, a user-defined function that will never be run on the z/OS platform because a developer runs SAS exclusively on Windows machines does not need to be engineered for that environment. To do so would waste developer resources.

Only a few pages in, and the critical importance of understanding software quality is already salient, including the use of nomenclature that describes specific characteristics of software quality. An understanding of this nomenclature benefits the discussion and documentation of software requirements and can help communicate to key stakeholders the many ways that user-defined functions provide value. Software requirements, after all, should drive the design and development of user-defined functions, communicate why a callable software module is needed, and also why a noncallable solution will not suffice. The next sections continue the discussion on software quality and provide a framework for discussing the benefits and value of user-defined functions.

The Many Facets of Software Quality and Performance

Software quality comprises a mix of both functionality and performance. If software aims to provide some algorithmic calculation but fails to generate the correct result, it can be said to lack quality because it does not produce the required functionality. But if the same software instead produces the correct result yet takes too long to compute (or hogs system resources), it also can be said to lack quality because it fails to deliver the required performance. In this vein, software requirements should convey both functional and performance requirements that specify not only what software must do but also how (or how well) it should do it.

This chapter began with the somewhat radical assertion that SAS software can produce equivalent functionality with or without the use of user-defined functions. Why then should SAS practitioners invest time in mastering the FCMP procedure and the design of user-defined functions? Because user-defined functions improve the *performance* of software, and in so doing, improve software quality.

Software performance is sometimes misconstrued as narrowly describing only processing speed or software efficiency; however, these are but two of a score of characteristics that can describe software performance. More broadly, the Institute of Electrical and Electronics Engineers (IEEE) defines *performance* as “the measurable criterion that identifies a quality attribute of a function

or how well a functional requirement must be accomplished" (IEEE 2005). And "software quality attributes" comprise *external software quality* and *internal software quality*. External software quality includes software characteristics such as speed, efficiency, reliability, and robustness that can be observed (and often measured) as software executes. Internal software quality, conversely, describes performance that cannot be assessed by running software—you must pry open a program and inspect its code to determine whether it is modular, readable, or reusable.

ISO defines an *internal measure of software quality* as a "measure of the degree to which a set of static attributes of a software product satisfies stated and implied needs for the software product to be used under specified conditions" (ISO/IEC 2014). ISO further clarifies that "Static attributes include those that relate to the software architecture, structure and its components. Static attributes can be verified by review, inspection, simulation, or automated tools." Thus, user-defined functions improve software performance by increasing the internal quality of the software, as measured by static quality attributes such as modularity, maintainability, reusability, and configurability. Internal software quality is often referred to as *static performance*, and external software quality as *dynamic performance*—the distinction representing whether software must be running or not to assess a particular quality attribute.

To bring this discussion full circle, SAS user-defined functions rarely make your software run faster or more efficiently. However, user-defined functions do improve a developer's ability to maintain and modify SAS software, as well as an end user's ability to use and interact with software. In this manner, user-defined functions can improve the quality of software, the quality of the development environment (that is, the experience of SAS practitioners writing SAS software), and the quality of the end-user experience. Several static performance attributes—including modularity, readability, configurability, reusability, maintainability, and integrity—are introduced in the next sections, as user-defined functions model these quality characteristics.

No respectable book about functions could begin without the ubiquitous example that converts between temperature scales. Program 1.5 demonstrates the FAHR_TO_CEL user-defined function that converts Fahrenheit to Celsius. It is referenced and refactored in the following sections to introduce software quality.

Program 1.5: Fahrenheit Conversion (FAHR_TO_CEL) User-Defined Function

```
* converts Fahrenheit temperature to Celsius;
proc fcmp outlib=work.funcs.num;
   function fahr_to_cel(f);
      c = (f - 32) * (5 / 9);
      return(c);
      endfunc;
quit;
```

Software Modularity

Software modularity describes the cleaving of software into discrete chunks of code to achieve the goal of *module independence*—the ability of a user to alter one module without affecting or

interfering with the functionality or performance of other modules. Modular software is often contrasted with *monolithic* software—*one stone*, in Greek—in which functionality is delivered through a single program file. Although modularity does tend to diminish software component size, breaking a monolithic program into bits does not, in and of itself, make it modular. Rather, truly modular software requires *loose coupling* of components, in which modules interact only where necessary, and only through prescribed communication channels.

In addition to displaying software independence, modular software is typically functionally discrete—that is, each module should have a singular focus and do *one and only one thing*. These two principal requirements for loose coupling and functional discretion are sometimes described as *low coupling with high cohesion* and contribute to module conciseness. Thus, the brevity typically demonstrated by software modules should not be considered to be a defining characteristic of software modularity, but rather a welcome consequence of functional discretion and loose coupling. It is this concise, modular design that lays the foundation for other software quality characteristics, as described in the following software quality sections.

A common method to promote software modularity is through *callable software modules*, in which a module's functionality is delivered by calling the module's name. Callable modules, which include both functions and subroutines, are introduced later in this chapter. For example, Program 1.5 demonstrates software modularity in that the FAHR_TO_CEL function does only one thing: converts Fahrenheit to Celsius. Moreover, FAHR_TO_CEL is segmented from other code—enclosed between the FUNCTION and ENDFUNC statements and encapsulated inside the FCMP procedure.

The following statement executed from a DATA step temporarily transfers program control to the FAHR_TO_CEL function when FAHR_TO_CEL is called:

```
celsius = fahr_to_cel(212)
```

However, the pinnacle of software modularity requires that callable modules not only be encapsulated but also be separated—that is, the calling module and called module should be maintained in different program files. This software design promotes software security and integrity because a user-defined function can be designed, developed, tested, and locked for read-only use prior to deployment to production. Thereafter, calling modules that use and reuse the function can be modified without risk of accidental alteration of the function itself. Moreover, reusability is promoted where functions are maintained in separate program files because multiple calling modules can call the same user-defined function.

For example, it should not be misconstrued that the prior call to FAHR_TO_CEL occurs in Program 1.5, in which FAHR_TO_CEL is defined; these represent two separate SAS program files. This distinction is made clearer in Chapter 2, in which user-defined functions are saved to a persistent SAS library rather than the ephemeral WORK library. Software modularity is discussed further in this chapter in the "Function Implementation" and "Function Invocation" sections, which explain that a function's implementation and its invocation generally should never occur in the same program file.

Software Readability

Software readability describes the ease with which software—including code, comments, and accompanying documentation—can be read and understood. Readability is especially important where software is expected to be maintained or modified by users who are not the original developers. Many aspects of code readability are not only language-dependent but also somewhat subjective. For example, indentation, line spacing, and other formatting can increase or decrease readability, as can variable-naming conventions or capitalization. But in many cases, style standardization is as important as the specific formatting or other conventions. Readability can also be improved through apt software organization and inline comments.

Notwithstanding the subjectivity that surrounds readability, some design practices do inarguably improve the ability of a developer to parse and understand code. Callable software modules (and the software modularity that they espouse) represent one such best practice. Readability of the *called module* is improved because the function is doing one and only one thing. For example, Program 1.5 converts a Fahrenheit temperature to Celsius, and nothing more, so its functionality is readily understood. The single code comment "transforms Fahrenheit temperature to Celsius" captures the high-level information that is required to call the function. *Fahrenheit* defines the input parameter, *Transforms* describes the functionality, and *Celsius* defines the return value or output.

Readability of the *calling program* is also improved through user-defined functions. Consider Program 1.6, which calls FAHR_TO_CEL to transform Temp1, and which transforms Temp2 using the equivalent hardcoded algorithm.

Program 1.6: Comparing FAHR_TO_CEL Function to Functionally Equivalent Hardcoded Transformation

```
data transformed;
   length temp_f temp_c1 temp_c2 8;
   temp_f = 212;
   temp_c1 = fahr_to_cel(temp_f);
   temp_c2 = (temp_f - 32) * (5 / 9);
run;
```

Inspecting the DATA step, it is clear that FAHR_TO_CEL is transforming the 212-degree boiling water from Fahrenheit to Celsius. Without having to recall junior high math, a developer can grasp this high-level functionality. However, the equivalent hardcoded transformation that initializes Temp_c2 is more complex both to write and to decipher. Now consider a more complex function that might perform advanced calculations comprising twenty lines of code. Despite this complexity, the function's invocation would still require only one SAS statement. But hardcode these twenty lines instead into a DATA step, and the high-level intent of the DATA step could be eclipsed by the code complexity. Thus, software design that modularizes functionality into user-defined functions enables developers to better comprehend high-level functionality without getting lost in the weeds.

Software Configurability

Software configurability describes the ease with which end users can interact with software to achieve dynamic functionality. Configurability is primarily engineered through function parameters, through which end users can alter a function's functionality by modifying one or more corresponding arguments when the function is called. Functions that are more configurable are able to meet the needs of more diverse users and more diverse use cases, and end users touting the benefits of a "highly flexibly function" are often describing a highly *configurable* function.

Program 1.5 declares a single parameter (F), which represents the temperature in Fahrenheit that is passed to the function. The resultant Celsius temperature is returned without rounding or truncation, so the following function call in which 211 is passed using the %SYSFUNC macro function returns quite a few superfluous 4s (99.4444444444444):

```
%put %sysfunc(fahr_to_cel(211));
```

The return value is accurate, although some users might prefer a rounded, more concise result. And where end-user preferences might differ, *configurability* can facilitate a single function that meets these diverse needs. The refactored FAHR_TO_CEL_RND function in Program 1.7 declares a second parameter (DEC), which defines the number of decimals of precision in the returned Celsius value.

Program 1.7: Adding a Parameter to Improve Configurability of a Function

```
* F - degrees Fahrenheit;
* DEC - decimals precision;
proc fcmp outlib=work.funcs.num;
   function fahr_to_cel_rnd(f, dec);
      c = (f - 32) * (5 / 9);
      rnd = 1 / (10**dec);
      return(round(c, rnd));
      endfunc;
quit;
```

This more configurable function, when called with two decimals of precision, now returns 99.44:

```
%put %sysfunc(fahr_to_cel_rnd(211, 2));
```

Developers and end users alike are more apt to favor configurable functions because functionality can be varied by modifying only the arguments within a function call, rather than having to modify the function's definition. In this way, configurability can facilitate more stable functions that require fewer modifications over time. Thus, rather than pursuing less sustainable *customization*, in which the needs of only one customer drive development, *configuration* instead aims to satisfy multiple, diverse customers using more flexible functions.

Software Reusability

Software reusability describes the ease with which software modules, including functions, can be reused—either in the same or future software products. Software reuse can dramatically increase the speed and efficiency with which software can be developed. For example, reuse of a user-defined function can rely on the previous design, development, testing, and documentation that has already been completed. In many cases, implementing an existing user-defined function within a new program can be drag-and-drop easy—name the location of the function using the CMPLIB system option, and call the function from the DATA step!

From a SAS practitioner's perspective, software reuse is arguably the primary rationale for mastering the FCMP procedure. We cannot maximize our productivity without embracing software reuse. And the modularity, callability, readability, and configurability discussed in the previous sections directly contribute to the likelihood that a user-defined function can and will be reused.

Modularity drives reuse because independent modules are disconnected. The requirement that modular software be loosely coupled means that a well-formed module often can be plucked from its original usage and reused elsewhere without adversity. *Callable modules*, including functions, further spur reuse because they can be invoked simply by calling the function name. Thus, productivity is radically improved when a 30-line function can be effortlessly included in your DATA step using a one-line function call.

Readability drives reuse because software modules that can be understood—especially at a high level—can be incorporated into software. In some cases, you might not understand *how* the function delivers its functionality, but as long as you understand *what* it delivers, you can still use the function. Finally, *configurability* encourages reuse because a function's functionality can be varied. Dynamic arguments produce dynamic results, and a more diverse array of users will find value in functions that can be readily configured.

Software Maintainability

Software maintainability describes the ease with which software can be maintained and modified, either by the developers who initially wrote the software or by separate developers tasked with software maintenance. Software that can be more readily modified reduces downtime and increases the speed and efficiency with which software can return to a functional state. Maintenance might be performed to correct a defect, improve performance, or extend functionality. But regardless of the driver, improved maintainability equates to higher *availability*—the "up" time that software is functioning and meeting business needs and requirements.

And software availability directly equates to dollars and cents—a language that product owners, customers, and other key stakeholders speak. The ability of user-defined functions to

improve availability through increased maintainability becomes a primary talking point when demonstrating the worth of user-defined functions to decision-makers.

Maintainability is principally driven by software modularity. Because user-defined functions are functionally discrete, concise, callable modules, they can be more readily understood and modified. Consider the extension of the FAHR_TO_CEL function (Program 1.5) to the FAHR_TO_CEL_RND function (Program 1.7), in which the added DEC parameter specifies the number of decimals in the return value. FAHR_TO_CEL didn't have much junk in its trunk, so comprehension of its functionality was straightforward, and this functionality could be extended easily by adding the DEC parameter.

Software reuse also improves maintainability because *reuse* denotes a module that is relied upon across software projects or across a team or enterprise. A user-defined function might be reused a dozen or more times by a team. If its functionality needs to be extended, this maintenance can be performed once, tested once, and redeployed once to alter functionality across all dozen instances in which the function is called. Without this reuse, maintenance is impeded because developers must modify separate programs individually, rather than altering one user-defined function. And again, from a business perspective, dysfunctional software—or software that is failing to meet business needs or failing to deliver business value—equates to lost revenue.

Software Integrity

Software integrity forms one leg of the confidentiality, integrity, and availability (CIA) security triad and describes the need to protect software against malicious, unauthorized, or inadvertent access or modification. Large, monolithic program files can be riskier because inevitable maintenance exposes the entire code to the risk of alteration. The cybersecurity principle of *least privilege* specifies that as few users as possible should have access to key infrastructure (such as code) and can mitigate risks to software integrity.

One best practice that maximizes software integrity is—say it with me—modular software design! Team leads or senior SAS practitioners can be charged with maintaining a library of reusable, user-defined functions. In so doing, they alone can be granted Edit permissions to modify the critical functions that underpin multiple software projects. Less experienced SAS users can be granted Read-Only permissions to user-defined functions, and thus can leverage these functions, but with the confidence that the function definitions and functionality cannot be modified.

Other methods that facilitate software integrity include formalized change management and release management policy and processes, attention to cybersecurity best practices, and implementation of security controls that can further mitigate or eliminate risk—all of which fall outside the scope of this text. However, restricting and delegating code access through modular software design is often a first step toward greater software security.

Functional Components and Organization

At a high level, successful function design for both built-in and user-defined functions requires developers to fulfill three objectives:

1. Discuss, define, and document the function's functionality and performance that will meet some business need and requirements.
2. Write code that delivers this functionality and performance.
3. Empower users to call the function to render its functionality.

These objectives represent separate components of callable software modules. They correspond to a function's specification, implementation, and invocation, respectively. The *specification* defines the function that software developers are building and subsequently instructs end users what has been built and how to interact with it. The *implementation* comprises the code—the meat of the function. The *invocation* represents the function call through which end users run the function. The next sections describe these three function components.

Function Specification

Before any code has been written, early in the software development life cycle (SDLC), a function typically begins with a specification—the "tech specs" that define software objectives, including the required functionality and performance. ISO defines a software *specification* as a "document that fully describes a design element or its interfaces in terms of requirements (functional, performance, constraints, and design characteristics) and the qualification conditions and procedures for each requirement" (International Organization for Standarization 2017). Technical requirements are crucial because they instruct developers *what* to build, as well as *when to stop* building, thus conveying the *definition of done* for each software component or product.

For example, when SAS software developers began conceptualizing the need for the built-in LOWCASE function, they undoubtedly described the function's intended *functionality*—conversion to lowercase—in its specification. However, they also would have defined the required *performance*, such as speed (for example, characters transformed per second) or interoperability (for example, operating environments in which LOWCASE should be compatible). Thus, during the design, development, and testing phases of the SDLC, the specification guides developers and helps ensure needs and requirements are delivered. And once a function passes testing, it can be released into production for use by end users during the operations and maintenance (O&M) phase.

During the O&M phase, the specification adopts a new role and conveys to users how to interact with a function. This user-focused specification (required to *run* software) will typically be far less technical than the corresponding technical specifications (required to *build* software). Thus, the specification available to end users will typically state what the function does (its functionality),

what input is required (its parameters), what output is produced (return values or return codes), as well as additional context or caveats that might assist users calling the function.

For example, as shown in Figure 1.1, the built-in LOWCASE function is masterfully described in the *SAS® 9.4 Functions and Call Routines: Reference, Fifth Edition* (SAS Institute Inc. 2020).

Figure 1.1: SAS LOWCASE Function Specification

LOWCASE Function

Converts all uppercase single-width English alphabet letters in an argument to lowercase.

Categories:	Character
	CAS
Restriction:	This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.
Note:	This function supports the VARCHAR type.

Table of Contents

Syntax

LOWCASE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the LOWCASE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The LOWCASE function copies the character argument, converts all uppercase single-width English alphabet letters to lowercase letters, and returns the altered value as a result.

The results of the LOWCASE function depend directly on the translation table that is in effect (see TRANTAB= System Option in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

A specification should contain sufficient information to enable users to call a function without inspecting the function's *implementation*—its underlying code. Note that the SAS specification for LOWCASE defines a single parameter (termed *argument* in SAS parlance) that must be a "character constant, variable, or expression," and also provides details about how the function can be used. Not depicted in Figure 1.1, the SAS LOWCASE specification also demonstrates examples of how to call LOWCASE within a DATA step.

User-defined functions developed using the FCMP procedure should also be accompanied by a specification that describes their functionality and usage to end users. Because one of the primary objectives of building user-defined functions is software reusability, a function might be developed by one SAS practitioner, yet shared among teammates and users throughout an organization, and persist far beyond the employment of the original developer. In these cases, a formal specification can best convey the functionality, usage, and caveats of a user-defined function. When specifications are absent, and especially when code is poorly documented or undocumented, users unfamiliar with a particular user-defined function are more likely to abandon its use or unnecessarily re-create its functionality—because they neither understand nor trust what the function does, and because they might have neither the time nor the skill set to parse through the function's implementation.

For example, Program 1.3 contained four inline comments that introduced the TINY user-defined function. Those comments effectively comprised a brief (yet viable) specification:

```
* converts character variable to lowercase;
* requires single character parameter <= 100 characters;
* no exception handling for arguments that exceed 100 characters;
* tested and intended for use ONLY in a Windows environment;
```

The specification conveys the high-level functionality (transformation to lowercase), the single required parameter (the character variable or value being transformed), the caveat that exception handling is absent, and the caveat that the function is intended only for a Windows environment. In many cases, this type of inline specification is sufficient. However, in some environments, an inline specification is insufficient (or disallowed), and an external specification (similar to that demonstrated in Figure 1.1) should accompany all user-defined functions.

Function Implementation

A function's implementation contains its code—it *implements* the objectives stated in the function's specification to deliver functionality to the user or process calling the function. ISO broadly defines an *implementation* as a "process of translating a design into hardware components, software components, or both" (International Organization for Standarization 2017). A function's implementation is commonly referred to as the function's *definition*, as it defines the functionality that is produced.

Built-in functions typically conceal their implementations. We know *what* a SAS built-in function does from reading its specification and observing its results, but not *how* it does it because

we cannot view the underlying C code. And with SAS investing billions to innovate and patent bleeding-edge technology to outpace its competitors, it is understandable why its source code remains copyrighted and concealed! User-defined functions similarly can be encrypted using the FCMP procedure ENCRYPT option, which facilitates delivering functionality without exposing proprietary methods to your user base. The ENCRYPT option is introduced and demonstrated in Chapter 2.

In general, however, SAS user-defined functions are unencrypted, and their code is exposed to the users calling them. Thus, the implementation of a user-defined function comprises the code between the FUNCTION and ENDFUNC statements, and the implementation of a user-defined subroutine comprises the code between the SUBROUTINE and ENDSUB statements. This openness facilitates a deeper understanding of functionality because SAS practitioners can inspect the code itself. It also facilitates maintainability because the function's implementation can be modified readily—either to alter or extend functionality, or to refactor the function to deliver increased performance. It is for this reason—the ease of access to the underlying code—that user-defined functions tend to be undocumented through external specifications. Many SAS practitioners instead rely on inline comments, as demonstrated in Program 1.3.

Function Invocation

The invocation is the third component of every function. It comprises the code that calls (invokes) the function. ISO defines an *invocation* as "the mapping of a parallel initiation of activities of an integral activity group that perform a distinct function and return to the initiating activity" (International Organization for Standarization 2017). More specifically, ISO defines a (function) *call* as "a transfer of control from one software module to another, usually with the implication that control will be returned to the calling module" (International Organization for Standarization 2017). In addition to transferring program control, the invocation also typically transfers arguments that are bound to parameters, as discussed later in this chapter.

SAS user-defined functions and subroutines arguably are most often called from the DATA step. When SAS encounters a function in a DATA step, in the blink of an eye, it transfers program control to the function, and when the function terminates, returns program control to the DATA step. However, numerous SAS procedures (and some SAS statements and SAS macro statements) also support calling functions. Some invocation methods limit functionality and other invocation methods expand functionality. Thus, function design will, in part, be driven by not only the function's intended functionality, but also the method(s) through which the function is intended to be invoked.

Functions, unlike subroutines, always return a value. For this reason, function calls but not subroutine calls often initialize variables through direct assignment within the DATA step. For example, as demonstrated in Program 1.4, the following statements call the LOWCASE built-in function and the TINY user-defined function, respectively. LOWCASE initializes Phrase1 to the LOWCASE return value, and TINY initializes Phrase2 to the TINY return value:

```
phrase1 = lowcase(phrase);
phrase2 = tiny(phrase);
```

Functions, unlike subroutines, can be called from the SQL procedure. For example, Program 1.8 demonstrates comparable SQL code that creates the equivalent Phrase1 and Phrase2 variables by calling LOWCASE and TINY, respectively.

Program 1.8: Calling Built-in and User-Defined Functions from PROC SQL

```
data text;
   phrase = 'SAS Applications Programming: A Gentle Introduction';
run;
proc sql noprint;
   create table lowered as
      select lowcase(phrase) as phrase1,
         tiny(phrase) as phrase2
            from text;
quit;
```

Note that user-defined functions that declare one or more array parameters (introduced in Chapter 3) cannot be called from the SQL procedure. This limitation occurs because a SAS array cannot first be declared in the SQL procedure prior to the function call as is required when these user-defined functions are called in a DATA step. Also note that user-defined *subroutines* cannot be called from the SQL procedure because the CALL statement required by subroutine invocations cannot be accommodated.

Functions, unlike subroutines, can also be called using the WHERE data set option, which can juxtapose a data set name within the DATA step or a SAS procedure. For example, in Program 1.9, the first DATA step creates two observations—the value of Phrase is title case in the first observation and lowercase in the second observation. Subsequently, the WHERE option is used in the SET statement of the DATA step and in the PRINT procedure, respectively, to select and print only the second observation.

Program 1.9: Calling User-Defined Functions Using the WHERE Data Set Option

```
data texts;
   phrase = 'SAS Applications Programming: A Gentle Introduction'; output;
   phrase = 'sas applications programming: a gentle introduction'; output;
run;

data select_lowered;
   set texts (where=(phrase=tiny(phrase)));
   put phrase=;
run;

proc print data=texts (where=(phrase=tiny(phrase)));
run;
```

In both usages, the WHERE clause evaluates that the second observation is already lowercase and selects only that observation.

As previously demonstrated, the %SYSFUNC macro function can also call a built-in or user-defined function. In the following statements, %SYSFUNC calls TINY, converts &PHRASE to lowercase, and prints the TINY return value to the log:

```
%let phrase = SAS Applications Programming: A Gentle Introduction;
%put %sysfunc(tiny(&phrase));
```

Function calls, unlike subroutine calls, can be parenthetically nested inside of other function calls or subroutine calls, with the innermost expression executing first. For example, the following DATA step statement first converts Phrase to lowercase, then converts Phrase to uppercase, after which Phrase_upper is initialized to the uppercase representation of Phrase:

```
phrase_upper = upcase(lowcase(phrase));
```

Similarly, the %SYSFUNC macro function can be parenthetically nested inside of other %SYSFUNC calls, %SYSCALL calls, or macro function calls. For example, the following statements nest the TINY function call (executed via %SYSFUNC) inside the %LENGTH macro function call:

```
%let phrase = SAS Applications Programming: A Gentle Introduction;
%put %length(%sysfunc(tiny(&phrase)));
```

TINY first lowers the case of &PHRASE, after which %LENGTH evaluates the length of the TINY return value. Note that macro functions like %LENGTH do not require the %SYSFUNC wrapper, whereas DATA step functions like LOWCASE or TINY do require %SYSFUNC when called using the SAS macro language.

For this reason, when DATA step functions are nested inside of each other and called using the SAS macro language, each function call must be wrapped in a separate instance of %SYSFUNC. For example, note the two instances of %SYSFUNC in the following %PUT statement, in which LOWCASE is first called to lower the case of &PHRASE, after which UPCASE is called to raise the case of &PHRASE:

```
%let phrase = SAS Applications Programming: A Gentle Introduction;
%put %sysfunc(upcase(%sysfunc(lowcase(&phrase))));
```

Subroutines, on the other hand, do not return a value, so subroutine calls cannot initialize a variable through direct assignment. Neither can subroutines be used in SAS expressions.

Subroutine calls, unlike function calls, also must be prefaced by the CALL statement. For this reason, subroutine calls cannot be nested inside of function calls or other subroutine calls in either DATA step statements or the SAS macro language. For example, the following DATA step statement calls the SORTC built-in subroutine to sort two variables (Var1 and Var2) horizontally:

```
call sortc(var1, var2);
```

Similarly, when called from the SAS macro language, subroutine calls must include the %SYSCALL macro statement. Note that SAS macro variables referenced in a %SYSCALL statement must be declared prior to usage. For example, the following code declares and initializes &VAR1 and &VAR2, after which %SYSCALL calls the SORTC built-in subroutine to reorder the macro variables:

```
%global var1 var2;
%let var1 = bananas;
%let var2 = apples;
%syscall sortc(var1, var2);
%put &=var1 &=var2;
```

The log demonstrates that the values of &VAR1 and &VAR2 have been switched—that is, alphabetized by SORTC:

```
VAR1=apples VAR2=bananas
```

Both functions and subroutines can be called from the COMPUTE block of the REPORT procedure, as demonstrated in Chapter 9. User-defined functions, as opposed to subroutines, can also be called from the FORMAT procedure by specifying the function name in the OTHER option, as also demonstrated in Chapter 9.

Finally, user-defined functions and subroutines can be called through, in addition to the preceding methods, an ever-increasing number of procedures, many of which leverage SAS Viya, SAS Cloud Analytic Services (CAS), and SAS LASR Analytic Server. Although not discussed in this text, the following procedures support various aspects of FCMP functionality and should be further explored:

- PROC CALIS
- PROC DS2
- PROC FORMAT
- PROC GA
- PROC GENMOD
- PROC GLIMMIX
- PROC IML
- PROC OPTMODEL
- PROC PHREG
- PROC MCMC
- PROC MODEL
- PROC MONTE
- PROC NLIN
- PROC NLMIXED
- PROC NLP
- PROC OPTMODEL
- PROC OPTLSO
- PROC QUANTREG

- SAS Risk Dimensions procedures
- PROC SEVERITY
- PROC SIMILARITY
- PROC SURVEYPHREG
- PROC SVM
- PROC TMODEL
- PROC TRANASSIGN
- PROC VARMAX

(See also, https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0pio2crltpr35n1ny010zrfbvc9.htm.)

Function Nomenclature

The preceding introduction to software quality and performance characteristics conveyed the importance of leveraging user-defined functions in software design, as well as how functions can improve specific aspects of software quality. But first, an introduction to quality-related nomenclature was required so that software quality characteristics could be defined and discussed. Similarly, any introduction to user-defined functions and function design is bolstered by defining programming-language-agnostic, function-related nomenclature.

The remainder of this chapter introduces function-related concepts. Calling modules, callable modules, and called modules are defined and differentiated, as are three types of SAS callable software modules—procedures, functions, and subroutines. Parameters and arguments are defined and differentiated, which aid in communicating *to* a function call, as are return values and return codes, which aid in communicating *from* a function call. Finally, built-in and user-defined functions are contrasted.

Calling Module, Callable Module, and Called Module

As defined previously, an *invocation* or *call* temporarily transfers program control—but transfers *from what*, and transfers *to what*? The *calling module* or *calling program* represents the code in which a function call occurs. For this reason, the calling module is sometimes referenced as the *parent*. For example, when the TINY function is called from the DATA step in Program 1.4, the DATA step is the calling module. And when TINY is called from the SQL procedure in Program 1.8, the SQL procedure is the calling module. The calling module transfers not only program control but also arguments (variable inputs) to the called module, and this communication is essential in enabling function flexibility.

A *callable module*, conversely, is a module executed by invoking its name. All functions and subroutines are callable modules. When a specific callable module is called, it is sometimes referenced as the *called module* to distinguish that it was, in fact, called—rather than merely having the capability to be called. SAS built-in procedures, functions, and subroutines also represent callable modules, as they are always invoked by calling their names.

To promote software modularity, a callable module nearly always should be saved as a separate program file apart from the calling module(s). Yes, during initial development, debugging, and testing of user-defined functions, it is common to both create and call a function in the same program file. However, production software typically demands that called and calling modules be separated so that they can be independently maintained. For example, once a user-defined function has been perfected and is in production, myriad programs and processes might separately use and reuse that same function, and each of those calling modules should *reference*—yet never *repeat*—that function's implementation (that is, its definition within the FCMP procedure).

Functions Versus Procedures

Procedures are commonplace within Base SAS. We use them to sort data sets (PROC SORT), analyze data (PROC MEANS), generate reports (PROC REPORT), and for myriad other actions. Procedures typically operate on entire data sets by evaluating, transforming, or representing those data. ISO defines a *procedure* as "a routine that does not return a value" (International Organization for Standardization and International Electrotechnical Commission 2012). Rather than returning a value, as a function does, a procedure typically generates output that describes a data object or modifies one or more data objects such as SAS data sets.

For example, the DATA step in Program 1.10 creates an unordered list of random numbers that ranges from 0 to 99, after which the SORT procedure orders these observations in ascending order.

Program 1.10: SORT Procedure to Order 100 Observations

```
data long (drop=i);
   length num 8;
   call streaminit(123);
   do i = 1 to 100;
      num = int(rand('uniform')*100);
      output;
      end;
run;

proc sort data=long out=long_sorted;
   by num;
run;
```

Whereas most SAS procedures operate on entire data sets, functions and subroutines typically operate on or within one observation. For example, the SORT *function* orders variables within an observation, whereas the SORT *procedure* orders observations within a data set. For this reason, the SORT function is sometimes anecdotally referred to as a *horizontal sort*, and the SORT procedure as a *vertical sort*.

Program 1.11 initializes 100 variables (Num1 to Num100) to random integers between 0 and 99, after which the SORT function subsequently reorders these values.

Program 1.11: SORT Function Orders 100 Values

```
data short (drop=rc);
   array num 8 num1 - num100;
   call streaminit(123);
   do over num;
      num = int(rand('uniform')*100);
      end;
   rc = sort(of num[*]);
run;
```

The SORT function generates a return code that reflects the completion status of the function—1 for success or 0 for failure. In this example, the RC variable is initialized but is unused, as it is unlikely that SORT will fail.

Despite the oversimplified distinction that *procedures operate on data sets,* whereas *functions operate on observations*, exceptions to this rule abound. As discussed, the OPEN built-in function opens a read-only stream to an entire data set, and CLOSE similarly closes the stream. The RUN_MACRO and RUN_SASFILE built-in functions, both of which are supported only within the FCMP procedure, also flout this rule. RUN_MACRO, for example, enables a SAS macro, DATA step, or SAS procedure to execute from inside a user-defined function. That is, FCMP enables mind-bending acrobatics such as DATA steps that run inside other DATA steps, as showcased in Chapters 5 and 6!

Functions Versus Subroutines

Having defined *functions* (in part) as callable software modules that "return a single value," let's upend the applecart and introduce *subroutines*—another callable software component and kissing cousin of functions. Throughout SAS literature and documentation, subroutines are rather confusingly referred to as *functions*, *routines, CALL routines*, *call subroutines*, *subroutine procedures*, and *subprograms*. Within SAS documentation, subroutines are sometimes defined as a SAS component wholly apart from functions, and at other times, a subordinate construct and class of function. For example, SAS documentation defines a *subroutine* as "a special type of function where return values are optional" (SAS Institute Inc. 2020). This SAS documentation furthermore differentiates that "functions and CALL routines have the same form, except CALL routines do not return a value, and CALL routines can modify their parameters." All of this ambiguity requires a bit more precision.

To be clear, both functions and subroutines are callable software modules, and the only distinction lies in that functions always return a value and subroutines never return a value. It is because of this return value that functions can initialize a variable through direct assignment, whereas subroutines cannot. However, both functions and subroutines can modify arguments passed to them when those arguments are specified by the OUTARGS statement, as discussed in Chapter 2.

Consider two built-in callable modules—the SORT function and the SORT subroutine (sometimes referred to as *CALL SORT*). Each module provides similar functionality, although through different methods, as demonstrated in Program 1.12.

Program 1.12: Comparison of SORT Function and SORT Subroutine

```
data sorted (drop=rc);
   a = 5;
   b = 15;
   c = 10;
   call sort(a, b, c);
   put a= b= c=;
   x = 5;
   y = 15;
   z = 10;
   rc = sort(x, y, z);
   put x= y= z=;
run;
```

The log demonstrates that both the A-B-C and the X-Y-Z series have been sorted, the values of B and C have been exchanged, and the values of Y and Z have been exchanged:

```
a=5 b=10 c=15
x=5 y=10 z=15
```

The SORT *function* generates a return code whose value is initialized to RC, whereas the SORT *subroutine* must be preceded by the CALL statement and does not generate a return value or return code. The use of the CALL statement to *call* subroutines explains why subroutines are commonly referred to as *CALL routines*. However, all functions, subroutines, and procedures are *called*, which complicates this anecdotal usage.

Within this text, *functions* are consistently defined as "callable modules that return a value," and *subroutines* as "callable modules that do *not* return a value." However, although subroutines do not return a value, they are nevertheless expert communicators and capable of modifying one or more variables in the calling program. For example, as demonstrated in Program 1.12, the SORT subroutine modifies the B and C variables. Thus, subroutines can modify variables in the calling program *indirectly*—that is, through *indirect assignment*—whereas a function can modify a single variable through *direct assignment*, and multiple variables through *indirect assignment*. The OUTARGS statement enables indirect assignment in both user-defined functions and subroutines.

Parameters Versus Arguments

One of the primary jobs of function calls is to pass arguments (inputs) from the calling program to the called module. It is, after all, the variability of these inputs that spawns variability in the return value, output, or other outcome of the function. Some functions do not require input, as demonstrated in Chapter 2, although these use cases are uncommon.

Each *argument* passed to a function must first be declared inside the function as a *parameter*, which defines the data type (character versus numeric), dimensionality (scalar versus array), length, and other attributes. ISO defines an *argument* as a "constant, variable, or expression used

in a call to a software module to specify data or program elements to be passed to that module" (International Organization for Standarization 2017). ISO contrasts a *parameter* as a "constant, variable, or expression that is used to pass values between software modules" (International Organization for Standarization 2017). In some literature and programming languages, parameters are referred to as *formal parameters*, and arguments are referred to as *actual parameters*.

These terms—parameter and argument—are often conflated or used interchangeably, and their usage can also differ among programming languages. Within this text, however, *parameters* denote the variables that are declared within a function, and *arguments* denote the corresponding values passed during a function call. That is, parameters exist within a function's implementation (or definition), and arguments exist within the function's invocation (or call). Stated another way, all parameters have local scope inside a function, excepting those parameters specified by the OUTARGS statement, which have global scope (and are thus accessible to the calling program). This distinction is explained in the "Declaring Parameters" section of Chapter 2.

Return Values Versus Return Codes

Whereas parameters and arguments facilitate communication *to* a callable module, including both functions and subroutines, return values and return codes communicate *from* functions (but not subroutines) to the calling program. The distinction between *return values* and *return codes* is subtle yet important, as this nomenclature can differentiate how function results are used by software. Both return values and return codes represent the results generated by functions, but return codes are conceptualized as a specific type of return value. ISO defines a *return value* as the "value assigned to a parameter by a called module for access by the calling module" (International Organization for Standarization 2017).

Within the FCMP procedure, the RETURN statement returns a value to the calling program. Built-in SAS functions operate similarly and return a single return value. For example, when the LOWCASE function lowers the case of a character variable or value, the lowercase text represents the return value.

Return codes, on the other hand, are a subset of return values that communicate completion status or other performance metrics for a called module. Thus, return values are sometimes said to convey *data* from a function, whereas return codes convey *metadata*. ISO defines a *return code* as a "code used to influence the execution of a calling module following a return from a called module" (International Organization for Standarization 2017). Because return codes can describe the success or failure of a function's execution, they are commonly used in exception handling routines that detect and handle anomalous or adverse events or states.

For example, Program 1.13 uses the OPEN function to open the File_missing data set. Because the data set does not exist, OPEN returns a *return value* of 0. However, this return value is also a *return code* because it reflects the failed state of the OPEN invocation. By convention, the variable initialized by the OPEN return code is named DSID (data set ID).

Program 1.13: Exception Handling Dynamically Routes Program Flow Based on DSID Return Code

```
data _null_;
   dsid = open('file_missing');
   if dsid > 0 then do;
      * additional code to interact with opened data set;
      sorted = attrc(dsid, 'sortedby');
      put sorted=;
      end;
   else put 'file cannot be opened';
run;
```

After DSID is initialized to the return code of 0, DSID is subsequently evaluated by the IF statement. Because the exception (that is, the missing data set) is programmatically detected, the IF block does not execute. This exception handling, facilitated by the return code of the OPEN function, ensures that subsequent actions that would require an open file (such as the ATTRC function, to retrieve the list of variables by which a data set is sorted) are not executed. Had OPEN succeeded, DSID would have been initialized to a positive integer starting with 1, and the list of sort variables (had the data set been sorted) would have been printed to the log.

In SAS literature, it is commonplace to see return codes that are generated, yet never evaluated, such as when hash methods like DEFINEKEY or DEFINEDATA initialize return codes. However, where risk exists that a function like OPEN could fail, exception handling routines are favored, and the programmatic evaluation of return codes is considered a best practice.

Built-in Functions Versus User-Defined Functions

Built-in functions are provided as part of a software application or programming language and comprise the building blocks with which developers can engineer more complex functionality. As a language matures and expands over time, the quantity and variety of built-in functions increase as new functionality is incorporated.

For example, SAS 9.4M6 introduced numerous "Git" functions such as GITFN_COMMIT, GITFN_PULL, and GITFN_PUSH for use with Git repositories like GitHub. The incorporation of these built-in functions into Base SAS *extends* the SAS programming language by increasing its out-of-the-box capabilities. The *SAS® 9.4 Functions and CALL Routines: Reference, Fifth Edition* describes hundreds of built-in functions and subroutines that span a variety of categories, including mathematical, statistical, character, date and time, file input/output (I/O), and other areas (SAS Institute Inc. 2022).

SAS built-in functions are written in C, a third-generation language (3GL) with more direct access to memory operations and other lower-level system functionality. Built-in functions are also tested rigorously to ensure they are robust to the various ways that they might be used or misused and to optimize their performance and efficiency. Finally, SAS built-in functions are documented thoroughly through SAS technical specifications that describe their syntax, usage,

and caveats. Thus, when first conceptualizing whether to design a user-defined function in any language, always first exhaust language documentation to ensure that a sufficient built-in function does not already perform the needed functionality.

User-defined functions, conversely, are created by users—SAS practitioners like you and me who build SAS software. SAS user-defined functions and subroutines are defined using the FCMP procedure and can be invoked through both the SAS language and the SAS macro language. User-defined functions, like their built-in counterparts, similarly extend a programming language by defining functionality not otherwise available through built-in functions. In doing so, user-defined functions increase the quality of not only software but also the software development environment, with the objectives of reusability and maintainability making the work of SAS practitioners more productive and pleasant.

As developers shift from being function *users* to becoming function *creators*, we can build better functions by modeling the best practices evinced by built-in functions, including not only functionality but also function performance, communication, and documentation. Yes, when you're tasked to design a function that transforms Fahrenheit to Celsius, you must prioritize functionality—getting the calculation right. However, also important is how the function uses resources, and whether it does so smartly and efficiently. Communication is also key, especially how the function should alert or respond to missing, atypical, or invalid data, and where and how notes, warnings, or runtime errors should be conveyed to users. Documentation, too, should succinctly describe to end users how a user-defined function should be called, and perhaps how it should not. Each of these design objectives can be pursued by observing and mimicking built-in functions and their behavior.

Conclusion

This chapter introduced functions and subroutines within the SAS language, including both SAS built-in functions and user-defined functions built using the FCMP procedure. With functions defined, the business case was made for why user-defined functions should be incorporated into SAS software, and how user-defined functions can facilitate and improve specific characteristics of software quality, such as maintainability, modularity, reusability, readability, and integrity. Function components were defined and demonstrated, including the specification, implementation, and invocation. Various methods of calling user-defined functions and subroutines were discussed. Finally, functional nomenclature was introduced, such as the distinction among calling, callable, and called modules; among procedures, functions, and subroutines; between parameters and arguments; and between return values and return codes.