SUBROUTINES

Logically, a subroutine is a computational abstraction. *Physically,* it is a way of gathering together the individual decisions and calculations necessary to perform a particular algorithm (e.g., bubble sort, or trigonometric sine) and packaging the code in a manner that it can be invoked as part of the execution of a larger program. Each subroutine has a name, or handle, by which it can be referenced and, optionally, a set of parameters, through which input data can be passed to the subroutine and computed results can be returned. Together, the subroutine name and the parameter list are known as the "signature" or interface of the subroutine (see APPLICATION PROGRAM INTERFACES).

Conceptually a subroutine (or subprogram) can be viewed as part of a larger program, where the "main" routine "calls" a series of subroutines, which themselves may call other subroutines. This hierarchy of subroutine invocations is known as a "call tree" and defines a computer program's control flow graph. As programming languages and software development methodologies evolved the concept of a subroutine has remained the same, although terminology used to describe it has varied. Other terms for a subroutine include: subprogram, function, procedure, operation, or method.

Historically, Maurice Wilkes invented the subroutine in 1949 as part of the EDSAC Project (the first stored-program computer). EDSAC used punched-paper tapes to represent the instructions and data that were to be loaded into the computer for execution. Developers realized that these paper tapes could be "reused" by loading different combinations of the same codes into the computer, with different data. The first subroutine library was a rack of paper tapes.

The early motivation for subroutine creation was to avoid wasteful usage of resources. It was not only wasteful to have two programmers write the same code twice, but it was also extremely wasteful to have duplicate copies of the same code occupying the limited amount of memory that was available on computers at the time. Parameters were added to subroutines to make them more "reusable," in that the parameters were used not only to pass data into the subroutine to be operated upon and subsequently to have calculated results returned back, but also to control the execution through runtime "options" (see SOFTWARE REUSABILITY). As will be discussed later, programming languages added different types of parameters and different forms of parameter passing, for reasons of efficiency, flexibility, and later, for reasons of reliability. Finally, further early motivation for subroutines came from the fact that their source code could be separately compiled (or assembled). The resulting object code could be stored in a subroutine library, then other programmers would not only not have to write the same code, but they also wouldn't have to waste their limited batch-job mainframe execution time by having to compile (and debug) it. They could just link the subroutine's object code into their own program for subsequent execution.

While the savings in development time and code size from using subroutines is fairly obvious, there is also a widely recognized penalty. The invocation of a subroutine requires a "context switch" inside the computer. This takes valuable time and consumes additional resources because, when invoking a subroutine, the internal state of the machine must be preserved (e.g., register values), the locations and values of the actual parameters must be specified, and the address of the subroutine must be branched to, with the return address somehow saved. Depending on the architecture of the computer, these operations were supported in different manners. Some computers simply provide two (or more) sets of registers and left their management up to the programmers. The GPR (general purpose register) based IBM mainframe, supported the "branch and link register (BALR) instruction to jump to a subroutine with the address of the start of the parameter list in one register and the address of the return address in another register. Similarly, the "load multiple" (LM) and "store multiple" (STM) instructions were used to save the current contents of the working registers and restore them. Stack-based machines, such as the Burroughs 5500, had a different, much simpler approach, as the parameter values and return address were placed on the stack. Finally, complex instruction set computers, such as the VAX PDP-11 from Digital Equipment Corporation had very elegant context switching instructions that took several milliseconds, requiring several dozen microinstructions to execute (see MICROPROGRAMMING).

To summarize, a parameterized routine has an increased chance of being reused because it addresses a wider domain of applicability. In addition, the use of subroutines results in a

- Reduction in coding effort and duplication of maintenance
- Reduction in load module size (if used more than once)
- Fewer coding bugs (i.e., bugs are eliminated early if software is used more frequently)
- Less detailed thinking required in the design effort
- Better program documentation

Because parameterized subroutines provide a higher level of abstraction with increased flexibility, adaptability and generality, they may be

- Harder to maintain
- Harder to understand the interaction and dependencies of parameters, especially if the subroutine is heavily parameterized

Finally, depending on the degree of linguistic support and compilation maturity, the use of parameterized subroutines may result in

- Less run time efficiency of code due to extra branching and context switching overhead
- Larger load module size if dead code has not been eliminated
- Potential abuse if no type checking is done on actual/formal parameter pairs

SUBROUTINE INVOCATION

As high-level programming languages emerged, new terminology was introduced to distinguish special forms of subroutines. The two major types of subroutines are *procedures* and *functions.* Procedures and functions both have names (or labels) and can accept input arguments as parameters, but differ in the way they return values. Functions return, at most, a single value. They were originally intended to be used in mathematical expressions [e.g., A =: SQRT(B) + SQRT(C) ;]. In contrast to these "in-line" invocations of functions, procedures were intended to be "called," or, in the case of object-oriented languages, methods were intended to have "messages sent to."

SUBROUTINE PARAMETERS

Subroutines use parameters to pass and return information. The actual parameter values passed to a subroutine are referred to as "arguments." Subroutine parameters fall into three categories: (1) data, to be used as input (operand) or that are generated as output (e.g., math function); (2) control values (options) that affect the processing of a subroutine; and (3) operations that are used in the processing of a subroutine. Not all programming languages support all of these parameter types. The following example illustrates their use. Here, a sort subprogram, written in Ada, takes as input a list of data

(TheList) to be sorted. The subprogram outputs the sorted list (this is referred to as a "sort-in-place" approach). It also is passed a control variable (Ascending), to indicate if the list should be sorted in ascending or descending order. Finally, the implementation of the subprogram is parameterized so that it can process data of any type (Element), kept in a list of any length, using a partial order function "<".

```
generic
   type Element is limited private;
   type List is array (Integer range <>) of Element;
   with function "<" (Left, Right : in Element) return Boolean is <>;
procedure Sort_In_Place (TheList : in out List; Ascending : Boolean := true:);
```

Example 1: Generic Ada Sort Subprogram

In Ada, one needs to "instantiate" this subprogram before invoking it. That is, since the subprogram is "generic," one needs to supply actual parameter values for the generic parameters. For example:

```
type Vector is array (Integer range <>) of Element;
procedure Sort is new Sort_In_Place (
        Element => BattingAverages,
        List => Vector,
        "<" => "<");
```

Then, to invoke the subroutine, one could say:

```
Sort ( TheList => Yankees, Ascending => true );
```

One should note that since Ada supports positional and default parameter values, the following invocation is equivalent to the previous example:

```
Sort (Yankees);
```

This example illustrates several aspects of features certain programming languages provide in support of parameterizing subroutines. These include:

1. Typed parameters
2. Positional parameters
3. Named parameters
4. Default parameters
5. Overloading
6. Generic parameters

Early programming languages did not provide for any type checking at compile or run time; therefore one could pass a character string to a subroutine when it was expecting an integer. This often occurred when the order of actual parameters indicated the correspondence between actual and formal parameters. To reduce errors, type checking was added along with named parameters. This meant that software developers, by using by-name parameter passing, wouldn't have to remember the order of parameters, in addition to increasing the readability of the invocation. Default parameters provide the user with the ability to select from alternative options when invoking the subroutine, with the convenience of not cluttering the interface, when the default value is desired. A popular use is to include a debug flag in the interface to the subroutine, with a default value of "false." For example:

```
procedure DOIT (ToThis : in Thing; Debug : in Boolean :=
   false);
```

This procedure could be invoked as follows:

```
DOIT ( ToThis => MyThing, Debug => false);
```

or

```
DOIT ( ToThis => MyThing );
```

In languages without default parameters, sometimes operator overloading is supported. In this case, two or more subroutines can share the same name, if the compiler can distinguish them by their parameter types. For example, in the case above, without default parameters, one could define the following two procedures (the first one overloading the second):

```
procedure DOIT ( ToThis : in Thing; Debug : in boolean );
procedure DOIT ( ToThis : in Thing );
```

Finally, generic parameters provide a different, more powerful kind of adaptability and flexibility for a subroutine. Generic parameters signal the compiler to parameterize the generated code so that the resulting code template can be instantiated dynamically, at run time. This should be contrasted to a macro, or preprocessor that generates code at compile time. Generic subroutines can be used to create parameterized types.

SUBROUTINE MODIFIERS

As mentioned earlier, not all programming languages support all forms of parameters. For example, in the programming language PL/I, subroutine arguments may be an expression, statement label, constant, variable, or subroutine name. In the programming language Java, subroutine arguments may be an object of any type. Early forms of the programming language COBOL did not support any form of parameter passing. In COBOL all communication between the calling routine and the subroutine was achieved through "side-effects." That is, by convention, the calling routine placed the subroutine input arguments into certain "global" variables, and the subroutine placed the generated results in other predefined locations that were in the scope of both routines. The "scope" of a subroutine is a key issue, visibility to data and other subroutines was either implicit (in the case of "built-in" subroutines in FORTRAN or PL/I) or explicit (Java class libraries). With operator overloading and overriding through inheritance, it oftentimes became difficult for software developers, as well as the compiler to determine which implementation of a subroutine was being invoked at any point in time. To simplify this problem, subprogram modifiers were introduced. For exam-

ple, in Java, a method can be marked "final, public, private, protected, abstract, static, native, and/or synchronized," with certain allowable combinations (e.g., public final static synchronized). Finally, certain programming languages required the use of subprogram modifiers to help the compiler optimize code. For example, the programming language PL/I required that all recursive functions be identified as "RECURSIVE."

PARAMETER-PASSING CONVENTIONS

The following approaches to parameter passing have appeared in various programming languages:

1. *Call By Value.* This is the most intuitive parameter-passing mode, from a functional programming perspective. All parameters are "input only" and may not be modified by the subroutine. In effect, a temporary copy of the value of the actual parameter is made, and the address of this copy is passed to the subroutine. This is the only practical method of passing parameters, when the actual parameter is an expression (e.g., $X + Y$). On the negative side from a performance perspective, the call by value parameter-passing mode can be very resource intensive for passing arrays, large data structures, or complex objects because the contents of the entire array must be copied into a temporary area in memory. This is the default parameter-passing method in Pascal and the only parameter passing mechanism in C.
2. *Call By Reference.* This is the simplest and most direct method of passing parameters to a subroutine. It supports both input and output parameters. In this approach, the address of the actual parameter, or address of a temporary storage location where the result of an expression, is passed to the subroutine, which then uses the address to indirectly access the value or write the results. This parameter-passing method is the (only) parameter-passing mechanism in the programming language FORTRAN. In the programming language PASCAL, the programmer must explicitly use the modifier "VAR" to force this parameter-passing mode. This method of parameter passing is particularly efficient when the parameters being passed are large structures, because only a single address needs to be passed in order to provide access to the entire contents of the large structure.
3. *Call By Name.* This parameter-passing technique was originally developed as part of early assembler language macroprocessor pages and later supported by the programming language Algol. It allows for symbolic manipulation through late binding of the expression that forms the parameter (called "thunks"). To support "call by name" parameter passing, the compiler must create a special subroutine that gets executed each time the parameter is referenced inside the subroutine. Because the value of each variable in an expression used as an actual in the parameter may change between each reference in subroutine, the results of the parameter reference can possibly change, making for an interesting side effect.
4. *Call By Copy.* This parameter-passing mode is similar to call by value, only the results are copied back (output) upon completion of the subroutine.

In conclusion, subroutines remain an essential feature of all programming languages. Compiler optimization techniques have reduced many of the processing inefficiencies in the past while preserving their labor saving and conceptual usefulness.

WILL TRACZ
Lockheed Martin Federal Systems

SUBROUTINES. See MACROS.