Functions or Subprograms (Subroutines)

In computer programming language, a subroutine is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.

In different programming languages a subroutine may be called a procedure, a function, a routine, a method, or a subprogram. The generic term callable unit is sometimes used.

As the name subprogram suggests, a subroutine behaves in much the same way as a computer program that is used as one step in a larger program or another subprogram. A subroutine is often coded so that it can be started (called) several times and/or from several places during one execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the call once the subroutine's task is done.

Subroutines are a powerful programming tool, and the syntax of many programming languages includes support for writing and using them. Judicious use of subroutines (for example, through the structured programming approach) will often substantially reduce the cost of developing and maintaining a large program, while increasing its quality and reliability. Subroutines, often collected into libraries, are an important mechanism for sharing and trading software. The discipline of object-oriented programming is based on objects and methods (which are subroutines attached to these objects or object classes).

History

In the (very) early assemblers, subroutine support was limited. Subroutines were not explicitly separated from each other or from the main program, and indeed the source code of a subroutine could be interspersed with that of other subprograms. Some assemblers would offer predefined macros to generate the call and return sequences. Later assemblers (1960s) had much more sophisticated support for both in-line and separately assembled subroutines that could be linked together.

Self-modifying code

The first use of subprograms was on early computers that were programmed in machine code or assembly language, and did not have a specific call instruction. On those computers, each subroutine call had to be implemented as a sequence of lower level machine instructions that relied on self-modifying code. By replacing the operand of a branch instruction at the end of the procedure's body, execution could then be returned to the proper location (designated by the return address) in the calling program (usually just after the instruction that jumped into the subroutine).

Main concepts

PDF-XChange
Click to buy NOW!
www.tracker-software.com

The content of a subroutine is its body, the piece of program code that is executed when the subroutine is called or invoked.
A subroutine may be written so that it expects to obtain one or more data values from the calling program (its parameters or formal parameters). The calling program provides actual values for these parameters, called arguments. Different programming languages may use different conventions for passing arguments:

Convention	Description	Common use
Call by value	Argument is evaluated and copy of value is passed to subroutine	C, C++, Java (References to objects and arrays are also passed by value)
Call by reference	Reference to argument, typically its address is passed	C++, Fortran, PL/1
Call by result	Parameter value is copied back to argument on return from the subroutine	Ada OUT parameters
Call by value-result	Parameter value is copied back on entry to the subroutine and again on return	Algol
Call by name	Like a macro – replace the parameters with the unevaluated argument expressions	Algol, Scala
Call by constant value	Like call by value except that the parameter is treated as a constant	PL/I NONASSIGNABLE parameters, Ada IN parameters

Language support

Programming languages usually include specific constructs to:

- delimit the part of the program (body) that makes up the subroutine
- assign an identifier (name) to the subroutine
- specify the names and/or data types of its parameters and/or return values
- provide a private naming scope for its temporary variables
- identify variables outside the subroutine that are accessible within it
- call the subroutine
- provide values to its parameters
- specify the return values from within its body
- return to the calling program
- dispose of the values returned by a call
- handle any exceptional conditions encountered during the call

- package subroutines into a module, library, object, class, etc.

Some programming languages, such as Visual Basic .NET, Pascal, Fortran, and Ada, distinguish between functions or function subprograms, which provide an explicit return value to the calling program, and subroutines or procedures, which do not. In those languages, function calls are normally embedded in expressions (e.g., a sqrt function may be called as y = z + sqrt(x)); whereas procedure calls behave syntactically as statements (e.g., a print procedure may be called as if x > 0 then print(x). Other languages, such as C and Lisp, do not make this distinction, and treat those terms as synonymous.

Advantages

The advantages of breaking a program into subroutines include:

- decomposing a complex programming task into simpler steps: this is one of the two main tools of structured programming, along with data structures
- reducing duplicate code within a program
- enabling reuse of code across multiple programs
- dividing a large programming task among various programmers, or various stages of a project
- hiding implementation details from users of the subroutine
- improving traceability, i.e. most languages offer ways to obtain the call trace which includes the names of the involved subroutines and perhaps even more information such as file names and line numbers; by not decomposing the code into subroutines, debugging would be impaired severely

Disadvantages

Invoking a subroutine (versus using in-line code) imposes some computational overhead in the call mechanism. The subroutine typically requires standard housekeeping code – both at entry to, and

exit from, the function (function prologue and epilogue – usually saving general purpose registers and return address as a minimum).

C and C++ examples

In the C and C++ programming languages, subprograms are termed functions (or member functions when associated with a class). These languages use the special keyword void to indicate that a function takes no parameters (especially in C) and/or does not return any value. Note that C/C++ functions can have side-effects, including modifying any variables which addresses are passed as parameters (i.e. passed by reference). Examples:

```
void function1(void) { /* some code */ }
```

The function does not return a value and has to be called as a stand-alone function, e.g., `function1();`

```
int function2(void)
{
    return 5;
}
```

This function returns a result (the number 5), and the call can be part of an expression, e.g.,

```
x + function2()
```

```
char function3(int number)
{
    char selection[] = {'S','M','T','W','F','T','S'};
    return selection[number];
}
```

This function converts a number between 0 to 6 into the initial letter of the corresponding day of the week, namely 0 to 'S', 1 to 'M', ..., 6 to 'S'. The result of calling it might be assigned to a variable, e.g., `num_day = function3(number);`.

```
void function4(int *pointer_to_var)
{
    (*pointer_to_var)++;
}
```

This function does not return a value but modifies the variable which address is passed as the parameter; it would be called with

"`function4(&variable_to_increment);`".

Visual Basic 6 Examples

In the Visual Basic 6 language, subprograms are termed functions or subs (or methodswhen associated with a class). Visual Basic 6 uses various terms called types to define what is being passed as a parameter. By default, an unspecified variable is registered as a variant type and can be passed as ByRef (default) or ByVal. Also, when a function or sub is declared, it is given a public, private, or friend designation, which determines whether it can be accessed outside the module and/or project that it was declared in.

- **By value [ByVal]** – a way of passing the value of an argument to a procedure instead of passing the address. This allows the procedure to access a copy of the variable. As a result, the variable's actual value can't be changed by the procedure to which it is passed.
- **By reference [ByRef]** – a way of passing the address of an argument to a procedure instead of passing the value. This allows the procedure to access the actual variable. As a result, the variable's actual value can be changed by the procedure to which it is passed. Unless otherwise specified, arguments are passed by reference.
- **Public** (optional) – indicates that the function procedure is accessible to all other procedures in all modules. If used in a module that contains an Option Private, the procedure is not available outside the project.
- **Private** (optional) – indicates that the function procedure is accessible only to other procedures in the module where it is declared.
- **Friend** (optional) – used only in a class module. Indicates that the Function procedure is visible throughout the project, but not visible to a controller of an instance of an object.

```
Private Function Function1()
    ' Some Code Here
End Function
```

The function does not return a value and has to be called as a stand-alone function, e.g., `Function1`

```
Private Function Function2() as Integer
    Function2 = 5
End Function
```

This function returns a result (the number 5), and the call can be part of an expression, e.g., `x + Function2()`

```
Private Function Function3(ByVal intValue as Integer) as String
    Dim strArray(6) as String
    strArray = Array("M", "T", "W", "T", "F", "S", "S")
    Function3 = strArray(intValue)
End Function
```

PDF-XChange
Click to buy NOW!
www.tracker-software.com

This function converts a number between 0 and 6 into the initial letter of the corresponding day of the week, namely 0 to 'M', 1 to 'T', ..., 6 to 'S'. The result of calling it might be assigned to a variable, e.g., `num_day = Function3(number)`.

```
Private Function Function4(ByRef intValue as Integer)
    intValue = intValue + 1
End Function
```

This function does not return a value but modifies the variable which address is passed as the parameter; it would be called with "`Function4(variable_to_increment)`".