# Lecture 07: Subprograms

Subprograms are the building blocks for structuring your program into readable components that are easy to follow. Most programming languages therefore use them. This lecture discusses the topic under the following subheadings:

- Fundamentals
- Design Issues
- Parameter Passing
- Recursion
- Overloaded Subprograms
- Overloaded Operators
- Generic Subprograms
- Co-routines
- Implementations
- Summary and Concluding Remarks

## 7.1 Fundamentals

**Definition:** A *subprogram* (also called a *subroutine*) is a component of a program that carries out a specific activity or set of related activities. With the exception of co-routines and concurrent calls, subprograms exhibit the following properties:

- The subprogram has a single entry point
- On execution of the call, the calling program component is suspended, and control is transferred to the called subprogram. When the subprogram completes, control is returned to the calling statement.
- The subprogram may be defined with parameters, in which case it must be called with corresponding arguments.

At the assembly and machine code level, subprogram calls are handled in a manner similar to interrupts. Housekeeping procedures associated with an interrupt include the following:

- Save *program counter* (PC) and central processing unit (CPU) state (registers) to a stack;
- Load PC with *interrupt vector* (the address of the interrupt statement);
- Execute interrupt instructions;
- Restore CPU state from the stack;
- Restore previous PC value.

A similar situation occurs in the case of a subprogram call:

- Save PC and relevant CPU registers to a stack;
- Load PC with subprogram address;
- Execute subprogram instruction(s);
- Restore CPU registers from the stack;
- Restore the previous PC value from the stack.

**Terminologies:** Sub-programs are called different things across programming languages. Common terminologies are procedure, function, method, routine. Following are some examples:

- In Java, subprograms are implemented as methods and inner classes
- In C, they are implemented as functions
- In C++: functions and/or class methods (which are also called member functions)
- Pascal, PL/SQL, and SQL: procedures and functions
- Ada and Visual Basic: procedures
- RPG: subroutines
- Python: functions

**Heading and Body:** A subprogram contains a heading and a body. The heading typically contains the name of the subprogram, the return type (the data type of the value returned when the subprogram terminates), parameters, and in some cases accessibility keywords.

**Parameters:** Parameters are placement holders for values that will be sent to the subprogram when it is called. There are different approaches to subprogram parameters as discussed below.

## 7.2 Design Issues

As mentioned in lecture 3 (section 3.2), different languages employ different structures for subprograms. Among the commonly known structures are the following:

- **Separate Subprograms:** Subprograms are written in separate files (as in Fortran).
- **Integrated Nested Subprograms:** Subprograms are part of the program file, and can be nested. Examples of languages that support this approach are Pascal, ALGOL, Fortran, JavaScript, PHP, RPG-400.
- **Integrated Independent Subprograms:** Subprograms are part of the program file, and cannot be nested. Examples of languages that support this approach are C, C++, Java, SmallTalk
- **Non-separated Subprograms:** Subprogram definitions are not separated from the main program (as in SNOBOL4).

Other design issues include:

- Nested or Independent
- Should subprograms be defined before usage (as in Pascal) of after (as in C-based languages)
- Should subprograms be defined ahead of the main-routine or after?
- Should there be support for subprogram overloading?
- Should there be support for generic subprogram?
- Are local variables statically or dynamically allocated?

Some languages (particularly C-based languages) have a special **main** subprogram that have to be included in programs that interact directly with end-users (called driver programs/classes); others such as Pascal, PL/SQL, etc. do not stipulate a specific name or structure for the main subprogram.

## 7.3 Parameter Handling

As you are aware, when a subprogram is called, the arguments supplied on the call are copied to their corresponding parameters in a positional manner. Care must be taken on the call to ensure that the supplied arguments are of the same data type as their corresponding parameters. Of interest also, is how the parameters are processed. Following are five common parameter-handling strategies.

**Pass-by-Value:** The argument(s) supplied on the call of the subprogram are copied to the subprogram's parameters for internal use in the subprogram. All major programming languages (C, C++, Java, C#, Pascal, Ada, etc.) support this kind of parameter passing.

**Pass-by-Result:** No value is transmitted to the subprogram on the call, but the arguments are supplied. The subprogram executes normally, but prior to transfer of control back to the calling statement, the data value(s) in the actual parameter(s) is/are copied back to the argument(s) used to call the subprogram. Languages that support this type of parameter passing include Pascal (via variable parameters), Ada (via out parameters), PL/SQL (via out parameters), and Fortran (via out parameters).

**Pass-by-Reference:** An access path to the address of each reference argument is provided on the call of the subprogram. As the subprogram executes, changes to its parameter(s) result in changes to the originally supplied argument(s) via the access path(s). Languages that support this approach include C (via pointers), C++ (via pointers and reference parameters), Java (on reference variables), Ada (via in-out parameters), and PHP and C# (via out-mode parameters). Python and Ruby employ a strategy called pass-by-assignment, which is similar to pass-by-reference.

## 7.3 Parameter Passing (continued)

**Pass-by-Name:** This is similar but more far-reaching than a call-by-reference. When a parameter is passed by name, the actual parameter is textually substituted for the corresponding formal parameter in all occurrences in the subprogram. For example, suppose that a subprogram needs to access several external files given different circumstances. Instead of hard-coding the file-names, or having lengthy code to determine which file to use on different scenarios, we could have a name-parameter for the filename, which is passed to the subprogram. The subprogram would then access whatever file is passed to it by name. This approach is supported in FoxPro. Other languages such as C++, Ada, and Java support name-parameters indirectly in their implementation of generic subprograms.

**Pass-by-Value-Result:** This approach is in effect, a combination of *pass-by-value*, and *pass-by-result*. On the call of the subprogram, the arguments are copied to corresponding parameter. During execution, the values of the parameters may change. On return, the parameter values are copied back to the original call arguments. For this reason, the approach is also referred to as *pass-by-copy* or pass-by-copy-restore. Languages such as Ada and PL/SQL support this strategy.

Since this is an important design consideration, it is critical that decisions on strategies to be implemented in a language be made early in the design stage. Also note that a language may exhibit multiple parameter-handling strategies.

**Type-Checking for Parameters**

In many early languages, parameter type checking was not done, thus leading to runtime errors. In most contemporary languages, this is done to ensure that arguments supplied on the call of a subprogram are consistent with the defined parameters for the subprogram. In languages that support data type conversion, this is done automatically for promotions; otherwise, casting is necessary.

## 7.4 Recursion

Recursion is the ability of a subprogram to invoke itself. Not all programming languages support this, but the ones that do are usually more flexible than the ones that do not. All contemporary programming languages support recursion. This was not always the case; languages such as COBOL, Fortran, and PL/1 did not always support recursion. Figure 7-1 provides examples of recursion in C++ and Java.

**Figure 7-1a: Recursive C++ Function**

```
// Iterative function for N!
double Fact (int n)
{
int x; double Result;
Result =1;                    // or Result = N
for (x =1; x<=n; x++)         // or for (x =N-1; x>=1; x--)
        Result = Result * x;  //  Result = Result * x
return Result;
}
```

```
// Recursive version of N!
double Fact (int n)
{
double Result;
if ((n==1) || (n == 0)) Result =1;
else Result = n * Fact (n-1);
return Result;
}
```

**Figure 7-1b: Recursive Java Methods**

```
// Iterative factorial method
public static double IterativeFactorial(int Number)
{
   double Fact = Number;
   for (int x = Number - 1; x >= 1; x--) Fact = Fact * x;
   return Fact;
} // End of Factorial Method
```

```
// Recursive factorial method
public static double Factorial(int Number)
{
   double Fact;
   if ((Number == 1) || (Number == 0)) Fact = 1;
   else Fact = Number * Factorial(Number - 1);
   return Fact;
} // End of Factorial Method
```

```
// Iterative String Reversal Method
public static String Reverse(String ThisString)
    {
     int y, z;
     char x;
     String Rev = " ";
     for (z = ThisString.length()-1; z >= 0; z--)
     { Rev += ThisString.charAt(z); }
     return Rev.trim();
    } // End of Reverse Method
```

```
// Recursive String Reversal Method
public static String Reverse(String ThisString)
{
   int Length = ThisString.length();
   String Rev;
   if (Length == 1)Rev = ThisString;
   else Rev = ThisString.substring(Length-1, Length) +
          Reverse(ThisString.substring(0, Length-1));
   return Rev;
} // End of Reverse Method
```

## 7.5 Overloaded Subprograms

An overloaded subprogram is a subprogram that has the same name as another, but differs in its parameter list and/or return type; the code for overloaded subprograms may also differ slightly. Subprogram overloading contributes to improved execution-time efficiency as well as *polymorphism* (to be discussed later). Examples of languages supporting subprogram overloading are Ada, C++, Java, and C#. Figure 7-2 provides an example of subprogram overloading in Java.

**Figure 7-2: Two Overloaded Java Constructors**

```
// Assume data items ID_Number, FirstName, LAstName, AddressLine1, AddressLine2, StateProv, Zip, E_Mail, Telephone
public CollegeMember() // Constructor
{
  ID_Number = DEFAULT_ID;
  FirstName = LastName = AddressLine1 = AddressLine2 = StateProv = Zip = E_Mail = " ";
}

public CollegeMember(CollegeMember Member) // Overloaded Constructor
{
  ID_Number = Member.ID_Number;
  LastName = Member.LastName;
  FirstName = Member.FirstName;
  AddressLine1 = Member.AddressLine1;
  AddressLine2 = Member.AddressLine2;
  StateProv = Member.StateProv;
  Zip = Member.Zip;
  E_Mail = Member.E_Mail;
  Telephone = Member.Telephone;
 } // End of constructor
```

## 7.6 Overloaded Operators

Operator overloading is the ability to redefine the meaning and behavior of an operator. Some languages facilitate operator overloading in a limited way for certain operators, but do not extend the courtesy to programmers. Java provides an example of this.

Some languages overload operators, and also extend the courtesy to the programmer. Examples of this group of languages are C++, Python, and Ruby. Figure 7-3 shows the C++ code for some overloaded operators.

**Figure 7-3: Overloaded Operator Functions in C++**

```
// The following C++ class allows for manipulation of 3D points
// (x, y, z coordinates).
// Operators +, -, *, and = are overloaded
class Point
{
  protected:
  int x, y, z;

  // Member functions
  Point(){x = y = z = 0;} // Constructor
  void Modify (Point P) { x = P.x; y = P.y;  z = P.z;} // Modify method
  void Print() {cout << x << “, “ << y << “, “ << z;} // Inline Print method

  // Prototypes for overloaded operator functions
  Point operator+ (Point P2); // P1 is implied
  Point operator- (Point P2); // P1 is implied
  Point operator* (Point P2); // P1 is implied
  Point operator= (Point P2); // P1 is implied
};

// Overloaded +
Point Point :: operator+(Point P2)
{
  Point Result;
  Result.x = x + P2.x;  Result.y = y + P2.y;  Result.z = z + P2.z;
  return Result;
}

// Overloaded -
Point Point :: operator-(Point P2)
{
  Point Result;
  Result.x = x - P2.x;  Result.y = y - P2.y;  Result.z = z - P2.z;
  return Result;
}
```

```
// Code continuation
// Overloaded *
Point Point :: operator*(Point P2)
{
  Point Result;
  Result.x = x * P2.x;  Result.y = y * P2.y;
  Result.z = z * P2.z;
  return Result;
}

// Overloaded =
Point Point :: operator=(Point P2)
{
  x =  P2.x;  y = P2.y;  z = P2.z;
  return this*;
}
```

## 7.7 Generic Subprogram

A generic subprogram is a polymorphic subprogram that has the ability to act on different data sets. Each time the subprogram is called, a specialization (instantiation) of itself is automatically created by the complier for the data set provided. Languages supporting generic subprograms: C++, Ada, Java, C#. Figure 7-4 provides an example in C++.

**Figure 7-4: Illustrating a Generic C++ Function to Process any of Three Files**

```
// Assume a college hierarchy consisting of classes CollegeMember, Employee, and Student.
// Each class has its constructors and polymorphic methods inputData(string inCategory, int x), modify(…), and toString()

// Global declarations
typedef CollegeMember* AffilList; typedef Employee* EmpList; typedef Student* StudList;

const string HEADING = "College Directory Prototype";
const string STUDENT = "Student"; const string EMPLOYEE = "Employee";  const string AFFILIATE = "Affiliate";
const int FILE_LIM = 20; const int DEFAULT_ID = 0;

// Function Prototypes
// Template function for input of students, employees, or affiliates
template <typename Ttype1, typename Ttype2> void InputMembers(string mCategory);
// Template function for listing students, employees, or affiliates
template <typename Ttype1,  typename Ttype2> void ListMembers(string mCategory);

// . . .

// Excerpt from main function to process the user's request
  switch (Option)
  {
   case 0: {ExitTime = true; break;}
   case 1: {InputMembers <AffilList, CollegeMember>(AFFILIATE); break;}
   case 2: {InputMembers <EmpList, Employee> (EMPLOYEE); break;}
   case 3: {InputMembers <StudList, Student>(STUDENT); break;}
   case 4: {ListMembers <AffilList, CollegeMember> (AFFILIATE); break;} // {ListMembers (AFFILIATE); break;}
   case 5: {ListMembers <EmpList, Employee> (EMPLOYEE); break;} // {ListMembers (EMPLOYEE); break;}
   case 6: {ListMembers <StudList, Student> (STUDENT); break;} // {ListMembers (STUDENT); break;}
   // ...
  };


// . . .
```

**Figure 7-4: Illustrating a Generic C++ Function to Process any of Three Files (continued)**

```
// The InputMembers Function
template <typename Ttype1, typename Ttype2> void InputMembers(string mCategory)
{
 // Note: Ttype1 represents AffilList, EmpList, or StudList
 // Ttype2 represents CollegeMember, Employee, or Student

Ttype1 ThisList;        int x, mLim;
 string lCategory = mCategory + "s";
 fstream MemberFile;
 string FileName = mCategory + ".dat";

 // Reset the number of members and allocate space for them
 cout << "Enter Number of " << lCategory << " to be Processed: "; cin >> mLim; getchar();
 //if (inMode == STUDENT) ThisList = new Student[mLim];
 ThisList = new Ttype2[mLim];

 for (x = 1; x <= mLim; x++)
 { ThisList[x-1] = Ttype2(); ThisList[x-1].InputData(mCategory, x); }

  // Write the array to the file
  MemberFile.open (FileName.c_str(), ios :: app | ios :: out | ios:: binary);
  MemberFile.write((char*) ThisList, sizeof(*ThisList));
  MemberFile.flush();
  MemberFile.close();
} // End of InputMembers Method

// ...
```

## 7.8 Co-routines

A co-routine is a subprogram that has multiple entry points which are also controlled by the co-routine. This is potentially very confusing, but also quite useful towards the support of concurrent processing (also called parallel processing). Among the contemporary languages available, only Lua supports co-routines. However, several languages support multi-threading .

## 7.9 Implementation

Implementation of subprograms is essentially facilitated by stacks. Each invocation of the subprogram represents a push onto a stack. Each return prompts a pop of the stack to return to the previous point of operation.

## 7.10 Summary and Concluding Remarks

Here is the summary of what has been covered in this lecture:

- A subprogram is a component of a program that carries out a specific activity or set of related activities. With the exception of co-routines and concurrent calls, a typical subprogram exhibits the following properties: single entry point; invoked by a call statement; upon completion, control is returned to the calling statement; may be defined with parameters that would require matching arguments on the call statement; may after execution, return a set of value(s) to the calling statement.
- For each language, there are housekeeping procedures that have to be followed when subprograms are called.
- Commonly known subprogram structures are: Separate Subprograms, Integrated Nested Subprograms, Integrated Independent Subprograms, and Non-separated Subprograms.
- Parameter handling strategies that have been identified include the following: Pass-by-Value, Pass-by-Result, Pass-by-Reference, Pass-by-Name, and Pass-by-Value-Result. A language may exhibit multiple strategies; since this is an important design issue, it must be decided early in the design stage of a language. When learning the language, it is important for the CS professional to recognize what strategies are being employed by the language.
- All contemporary programming languages support the principle of recursion. In learning a new language, it is important to observe how recursion is supported in the language.
- Some languages support programmer-initiated subprogram overloading, operator overloading and generic subprograms.
- Some languages support co-routines; this feature can be useful in parallel processing.

The features of subprogram overloading, operator overloading, and generic subprograms, if present in a programming language, often contribute to enhanced execution-time efficiency as well as the achievement of polymorphism. This latter principle of polymorphism, remains a fundamental tenet of object-oriented programming (OOP), and fill be further discussed in lecture 9. The next lecture discusses support for abstract data types.

## 7.11 Recommended Readings

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapter 9.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 9.