Subprograms

Learning Objective

This session will introduce students to the concept of process abstraction - specifically, subprograms (subroutines and functions). We will cover the reasons for using subprograms, as well as their syntax and semantics.

Overview

The earliest computer programs were a handful of instructions. Computer memory was very limited, placing severe restrictions on the size of programs. But memory started getting cheaper and cheaper, and programs started getting bigger and bigger. Soon, programs became too big to write and too big to comprehend. A means of breaking them up into manageable pieces needed to be found. The solution to the problem was the invention of subprograms.

What is a Subprogram, and why would you use one?

A subprogram is nothing more than a process abstraction. Think of a process that you do everyday - say, making coffee. What is the process for making coffee? Get out a filter. Place filter in holder. Get out coffee. Place two scoops (three on a bad morning) in filter. Close filter assembly. Fill tank with water. Push "start" button. When someone asks what you are doing, you don't reply "I'm getting out a filter. I'm placing the filter in the holder .." Instead, you reply "I'm making coffee." You have created a process abstraction, by packaging the process and naming it with a simple descriptor.

This is exactly what is done with programming language subprograms. One may have a piece of code to alphabetically sort peoples' names. The code may be a few dozen statements, each one accomplishing a step of the sorting process. It is not unlikely that there would be multiple places in a large program that require a name-sorting capability. The software engineer could simply "cut and paste" the sorting code, using a copy every time a sort is needed. But, it would be a lot easier to place the sorting code in one place, and when it is necessary to do a sort, to do it with a single command: SORT.

Subprograms generally fall into two categories:

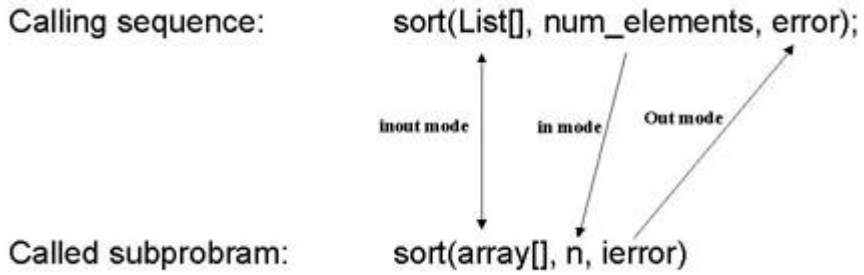
- ***procedures:*** used to execute a group of statements. They produce results in the calling program via shared and/or parameterized variables. FORTRAN procedures are called ***subroutines.***
- ***functions:*** used to determine and return a value that is used in an expression. Functions can also produce results in the calling program through *side effects*, which we'll look at shortly.

Parameter Passing

Parameter passing is the technique by which parameters are passed to and from subprograms. There are three categories of parameter passing:

- ***in mode:*** a formal parameter can receive (accept) data from an actual parameter, i.e., data can only be passed into the subprogram through the parameter.
- ***out mode:*** a formal parameter can transmit data to the actual parameter, i.e., data can only be passed out of the subprogram through the parameter.
- ***In-out mode:*** a formal parameter can receive or transmit data from/to an actual parameter, i.e., data can be passed either in or out of the subprogram through the parameter.

These three modes of parameter passing are illustrated here for SORT:



We've added a parameter ierror. The rationale for using these modes is as follows. LIST is the list of names, which goes into the subprogram unsorted, and returns through the parameter sorted. num-elements is the number of names in the list; there is no reason for the subprogram to change its value, so that parameter is in-mode. error is an indicator that something has gone wrong in the subprogram; therefore, it can be out-mode (you'd never pass an error flag into the subprogram).

The use of these mode improves the program's reliability. A coding mistake, e.g., trying to pass an error into the subprogram, would be caught at compilation time if the error parameter is declared as out-mode.

The next issue is how *data* is actually passed to a subprogram. There are five ways to do it:

- ***pass-by-value:*** This method is applicable to in-mode only, and is the default method in many languages. The value of the actual parameter is copied into the memory location of the formal parameter. The disadvantage of this method is that it is slow and takes more memory for the copy.
- ***pass-by-result:*** This method is applicable to out-mode. The value of the formal parameter is copied to the actual parameter memory location upon completion of the subprogram. This method can lead to *parameter collisions*.
- ***pass-by-value-result:*** This method is applicable to the inout mode. It's really a combination of pass-by-value and pass-by-result, and suffers their disadvantages.
- ***pass-by-reference:*** This method is applicable to the inout mode. It is somewhat similar to a global variable in that any changes to a formal parameter also change the actual parameter. In this mode, variable addresses, rather than values, are passed. This method

leads to the possibility of an effect known as ***aliasing***, where different variables mistakenly get treated as identical.

- ***pass-by-name:*** This is a rarely used method (e.g., ALGOL 60) applicable to inout mode. In this mode, the literal name of the parameter is passed, as opposed to a value or a memory location. Pass-by-name is a powerful construct in that functions, procedures, etc. can be passed as arguments. It pays for this flexibility in slow execution speed.

Overloaded and Generic Subprograms and Operators

Overloading operators and subprograms are basically the same idea: one has an operator (for instance, the addition {**+**} operator) or a subprogram (for instance, the SORT subprogram that we looked at earlier), but the operator or subprogram may have an additional capability to "morph" their functionality to meet different requirements. The same concept can be applied to subprograms. One might have a SORT program that sorts names alphabetically, but also could sort a list of integers into ascending order.

In short, the purpose of overloading is to extend the applicability of an operator or a subprogram to include new data types. You might wonder how the compiler knows which version to use (e.g., whether arithmetic addition or character concatenation) in a given situation. It does so by looking at the operator or subprogram in conjunction with the data types being operated on, and picks the appropriate version.

Generally speaking, operator overloading is a powerful capability that can dramatically increase program *writability*, but possibly at the expense of program *readability*. The readability may be impacted if the reader does not know the details of the overloading strategy. An inherent danger of user-defined overloaded operators/subprograms in in modular programs that are developed and compiled by different teams - do the teams all have the same understanding of the meaning of the new overloaded operators?

Despite these caveats, overloading is a powerful and useful concept, it is likely to be incorporated in more and more new languages.

Generic or **polymorphic** subprograms are a somewhat similar concept, in that they can accept different parameter types on different activations. Like an overloaded subprogram, one might use a generic subprogram that could automatically sort characters, floats, long or short ints, etc. Again, using the same subprogram name enhances code readability.

C++ and Ada are two languages that support generics. In Ada, generics are implemented with a ***generic*** construct that builds a ***template*** for the subprogram; the actual subprogram is ***instantiated*** (turned into actual code) when a specific call is made with specific type parameters. C++ implements generic subprograms with the ***template*** construct in a manner similar to Ada's.

Subprogram Compilation

One of the motivations for using subprograms that we discussed earlier is that they allow independent development of software modules by programming teams. There are also other advantages, e.g., compilation and recompilation times can be reduced (if a subprogram is modified, only that subprogram needs to be recompiled), and it makes software maintenance easier.

Separate compilation means that subprogram modules can be compiled at different times, but they are not independent in that type checking can still occur. Ada, FORTRAN 90, and Modula-2 provide this capability. The distinguishing feature of separate compilation is that the compiler must have knowledge of the other parts (modules) of the program.

Independent compilation means that separate program units (subprograms, etc.) are compiled without information about other program units. Languages that use independent compilation cannot check for type consistency between compiled units. C and FORTRAN 77 support independent compilation.

Some languages such as FORTRAN II and early versions of Pascal don't support either separate or independent compilation; therefore, they are not very useful languages for heavy-duty industrial applications, or large team-development efforts.

Functions

As noted previously, a ***function*** is a special case of a subprogram in which a value is returned. For example, `X = SQRT(2)` returns the square root of the parameter 2.

By their nature, function parameters should be ***in mode***, to prevent side effects from being transferred via parameters. Ada requires this, most other languages don't.

While most languages only allow simple (basic) types to be returned by a function, some languages allow complex structures to be returned via pointers, e.g., C and C++. Ada can return values of any type.

Access to Non-Local Environments

Usually, data gets passed to and from subprograms via parameters, but there are other ways to do it. ***Global variables*** are variables declared in a manner such that they are visible to all program units. Similarly, ***non-local variables*** are visible to a particular subprogram, but they are not defined in that subprogram.

FORTRAN provides a mechanism known as ***common blocks*** as a means of sharing data. Common blocks define memory that is shared between program units. Unfortunately, common blocks are inherently unreliable since they are prone to programmer error.

External declarations of variables allow data sharing between program units. Ada and Modula-2 allow specification of external modules whose data can be accessed.

Java and C++ can accommodate data sharing through the use of classes.

C uses the ***extern*** statement to specify variables that are declared outside the module.

In summary, the best technique (in terms of readability and reliability) to pass data to/from a subprogram is through its parameters, but other methods exist, and they are widely used in practice.

Coroutines

Coroutines are a somewhat different concept from subprograms. Whereas subprograms are subservient to the calling (main) program, coroutines are "near equals" to the calling program. Another difference is that coroutines may have multiple entry points.

Typically, coroutines will execute incrementally; the coroutine may start execution, then suspend, and then resume execution later. In that sense, they are analogous to multiprocessing in that processor resources are shared, but they give the appearance of concurrency. With that in mind, it's not surprising that coroutines find use in simulations, games, and other applications requiring concurrency.

Programming languages that support coroutines include SIMULA, BLISS, INTERLISP, and Modula-2.

In the grand scheme of things, coroutines are not in the mainstream.

Implementing Subprograms

Basically, we're interested in an overview of what must happen inside the computer when a subprogram is called, and when it's finished executing and control is returned to the main program.

These are the things that must happen when a subprogram is called:

- *There must be a mechanism for passing the parameters.* This is accomplished by having the compiler set aside a specific area of the computer's memory to hold the parameters.
- *Provision must be made for allocating memory for the subprogram's variables.* Typically, an area of memory is held in reserve, for use by subprograms when they are called. When the subprogram completes executing, the memory is released.
- *Provision must be made for saving the status of the calling program.* You will recall from Session 2 that CPUs have internal memory (registers) used to store instructions and data. A main (calling) program will be using those registers when a subprogram is called. Since the subprogram will want to use those same registers, the information in them must be saved (in regular memory), or it will be lost.
- *Provision must be made for enabling subprogram access to non-local variables.* We discussed this earlier.

- *Provision must be made to transfer control to the subprogram.* The subprogram instructions will occupy its own portion of memory. Control is transferred by jumping the instruction counter to the location of the first subprogram instruction.

When a subprogram completes execution, the above process is essentially reversed:

- Provision must be made for returning ***out-mode variables.***
- Provision must be made for releasing subprogram memory.
- Provision must be made for restoring the state of the calling program (i.e., restoring the registers to their original state)
- Control must be returned to the calling program.

The bottom line: calling subprograms is computationally intensive and expensive. Hence, the advantages of using subprograms (enhanced readability and maintainability, etc.) are offset by reduced performance. Hence, time-critical (e.g., real-time systems, graphic-intensive games) systems may have few if any subprograms.

Wrap Up

We've covered a lot of topics in this session: the concept of subprograms as process abstractions, actual versus formal parameters, functions versus procedures, modes of parameter passing, subprogram and operator overloading, coroutines, and how subprograms are implemented. Next session well move to the forefront of programming languages: abstract data types and object-oriented programming.

Abstract Data Types and Object-Oriented Programming

In the second lecture we looked at the early evolution of computer programming. We saw that early computers were built on the von Neumann model of fetch, execute, fetch, execute, ..., where instructions are fetched and then executed. As programming languages developed, they were naturally based on the von Neumann paradigm, where the focus is on *executing* instructions which in turn manipulate data. Such programming languages are said to be ***procedure oriented***.

As the software industry matured, a couple of things happened. Procedure-oriented programs got bigger and bigger, as did the data sets that they manipulated. The structure of the data itself also became more and more complicated. Programs and their associated data sets became too large for individual programmers to write and maintain. Some stop-gap advances helped a little, such as increased use of subprograms (which helped by introducing abstraction into the procedures), and increasing capability for programming languages to deal with abstract data types. Hence, the trend was towards greater levels of abstraction, and to shift from procedure-oriented

programming to data-oriented programming. This new approach began to be known as ***object-oriented*** programming (OOP).

This session will first look at data-abstraction concepts, and in particular, the idea of encapsulation. Then, we'll turn to the broader aspects of object-oriented programming.

Abstraction and Encapsulation

Turning to Webster for a definition of abstraction:

> *Formulation of an idea, as the qualities or properties of a thing, by mental separation from particular instances of material objects.*

Think of the abstraction "car": a metal box on four wheels with an engine for propulsion ... Using the single word "car" can immediately conjure an image in your mind of a complicated piece of machinery. This is also be defined as: "*... a view or representation of an entity that includes only the attributes of significance in a particular context."*

As mentioned in the Overview, modern programs are big: often hundreds of thousands, or even millions of lines of code. The only way to build and maintain programs of this size is through ***modularization***, in which subprograms with related functionality and their associated data are grouped together into modules. Hence, these modules accomplish yet another layer of abstraction.

Modularization accomplishes at least three goals:

- It simplifies program design and maintenance
- It allows for modular testing
- It reduces compilation/recompilation time.

Abstraction is an invaluable aid to modularization. We've seen that both data and procedures can be abstracted; data can be abstracted through the use of complex data types (e.g., arrays, records, etc.), and procedures can be abstracted through subprograms. The next step in this progression is to abstract *combined* data and the processes used to manipulate the data. This concept, called ***encapsulation,*** is the grouping of subprograms and the data they manipulate; as separately or independently compilable modules, encapsulations can achieve the three goals described above.

Encapsulation involves abstract data types. An abstract data type is an encapsulation of one specific data type, and the subprograms that operate on that type. Abstract data types are built using the "basic" data types, such as those that we covered earlier. For the purposes of this discussion, we'll use the terms 'abstract data types' and 'encapsulation' interchangably.

Note also that in a limited sense, the 'basic' data types (char, int, float, etc.) that we looked at earlier that are simple examples of abstract data types: i) they can only be manipulated with methods associated with the data type (i.e., certain arithmetic operators are associated with ints, and 2) their internal data representation is "hidden" (consider, for example, the internal representation of floats).

From a software-development perspective, one of the attractive features of encapsulation is the information-hiding aspect of the concept. There is no need for anyone other than the developer to know the details of what's going on inside the encapsulation. A user does not need to know what sort of data structure holds the data, or how the subprograms (Methods) are implemented. In theory, a user can get all required data and services via a well-defined interface (using the methods). If this were not the case, users might be tempted to access data using their own methods, if they knew the data's structure. If the developer, during the course of software maintenance, were to make changes to the data structures, the user's "home built" interface might stop working. Hence, the use of encapsulation is a step towards maintainable software.

Two more definitions: First, the encapsulation of specific data (e.g., an actual patient's name, height, weight, etc.), along with the encapsulation's methods, is called an ***object.*** Object-oriented programming is about how objects are created and manipulated.

Before moving on to object-oriented programming, we need to look at a few more definitions:

> ***object:*** *an object is a self-contained entity that consists of both data and procedures to manipulate the data.*
>
> ***class:*** *a category of objects. For example, there might be a class called shape that contains objects which are circles, rectangles, and triangles. The class defines all the common properties of the different objects that belong to it.*

'Object,' therefore, is just another term for an encapsulation.

Another widely heard term is "instance." An object is nothing more than an ***instance*** of a class.

Now, let's take a look at OOP.

Object-Oriented Programming

Object-oriented programming languages are becoming pervasive; they include C++, Smalltalk, Java, Ada 95, CLOS (an OO version of LISP), and Eiffel. OO programming started with SIMULA 67 (yes, over 30 years ago), and was developed further in Smalltalk 80.

A 'pure' OO programming language is one which supports:

- Abstract data types (i.e., encapsulation)
- Inheritance
- Dynamic binding

We have already discussed abstract data types in some length.

Inheritance also had its origins in the need to improve productivity in writing large software systems. The idea of software *reuse* became popular as a potential means for enhancing productivity; rather than write new code for every application, a team would attempt to reuse

previously written software. The problem with reuse was that existing modules are often "not quite right"

When a program sends a message to an object requesting the services of a method, the subclass of the object checks to see if it has a method with exactly the same name and parameters. If not, the parent class is examined. This continues up the ancestral chain until a match is found; if none is found, an error is declared.

Dynamic binding is the last of the 'big three' features of OO programming languages. *Dynamic* refers to a language's ability to associate (*bind*) a method to an object at run time (as opposed to compile time). Rather than the compiler determining which method to use when it compiles the code, it generates program code to make this determination at run time. Dynamic binding is also sometimes referred to as ***late binding***, for obvious reasons.

One of the downsides of OO programming languages is that their run-time efficiency (execution speed and memory usage) is less than that of non-OO imperative languages. This is because of the overhead associated with capabilities such as dynamic binding, and with the creation and bookkeeping associated with objects.

Wrap Up

Object-oriented programming, with its associated concepts of data abstraction (encapsulation), inheritance, and dynamic binding, has transformed the software industry. It has provided the machinery for creating large, complex programs using teams of developers, and it has made software reuse practical.

Next week, we'll look at two more powerful features of modern programming languages: concurrency and exception handling.