CS111 Introduction to Systematic Programming

More on Procedures -- Unit 9 Supplementary Hand-out

Subprograms

Collectively procedures and functions are known as **subprograms**. The advantages of procedurisation apply to all subprograms, that is both procedures and functions.

Naming Conventions for Subprograms

Procedure names are normally chosen to be **verb phrases** (or "doing" phrases) for example `Get`, `Put`, `ProduceFactors`, `OutputHeading` etc.

Function names are usually chosen to be **noun phrases** (or "naming" phrases), for example `MonthLength, Maximum`, `Minimum`, `Sin(e)`, `Cos(ine)`, `Log(arithm)` which, in a certain sense, name the value returned.

Names of predicates (or `Boolean`-valued functions) are usually predicate phrases, for example `IsLeapYear`, `IsPrimeNumber` etc..

Avoid procedure names that are too specific such as `DoThisAndDoThat` as they indicate that the procedure performs two or more tasks and suggest lack of cohesion. Consider splitting the procedure up into two separate procedures to perform the each task separately or perhaps choose a less specific name which suggests a unity of action. Thus for example `ProcessDate` is preferable to `InputAndOutputDate` and `CalculateAndOutputSalaryAndTaxInfo` would be better named as (say) `ProducePaySlip`.

Reasons for Using Subprograms

1. Avoids code duplication -- the same section of program code can be called from different parts of a program by calling the named subprogram.
2. Promotes code re-use -- utility procedures and functions can be collected together in library packages and imported into many programs, for example I/O procedures, maths libraries etc..
3. Improves program clarity provided procedures and functions have meaningful names -- top-down design with procedural decomposition.
4. As procedure call steps are separated from procedure definitions, there is a separation of what a procedure/function does from the details of how it does it. Hence program clarity is improved. At the head of each procedure/function definition there should be a brief comment describing WHAT IT DOES (but NOT HOW IT DOES IT) and documenting what parameters it requires. Detailed comments of HOW a procedure/function works should appear in the subprogram body not in its heading.
5. Ease of modification. A subprogram body can be modified (to correct bugs or improve the efficiency of an algorithm etc.) without altering the rest of the program code.

Advantages of Using Local Variables in Subprograms

1. Makes procedures and functions 'self-contained' and so promotes code re-use.
2. Improves program clarity as variables are defined near to where they are used.
3. Reduces the dangers of name clashes in large programs as each local variable can only be used within its own subprogram. Local variables in different subprograms are completely different entities even if they happen to have the same name.
4. Reduces the overall storage requirements of the program as storage is allocated for local variables only when the subprogram to which it belongs is active. When a subprogram 'returns' this storage is 'recycled' and is available for use by other subprograms.

Reasons for Using Parameters in Procedures and Functions

1. To enable information to be passed to a subprogram and so allow it to perform varying tasks whilst still being 'self-contained'.
2. To clearly specify the interface between a subprogram and its caller and so improve program clarity. By inspecting the parameters of a procedure we can see
   a) what information is being passed into the procedure from the caller via `IN` mode parameters.
   b) what information is being passed from the procedure to its caller via `OUT` mode parameters.
   c) what information is being passed into the procedure, modified and then passed back to the caller via `IN OUT` mode parameters.

No information should be passed to/from the procedure other than by the parameter passing mechanism as this reduces program clarity and can lead to obscure 'bugs'.

To illustrate the convenience of using parameters consider a hypothetical library procedure `Update` for modifying an integer value in some useful way. The first version works by updating a global variable `Value` whilst the second uses an `IN OUT` parameter of the same name. To use the first version of the procedure to update a variable `MyTotal` say we would need to do

```
   Value : Integer;     -- must remember to declare global variable
   MyTotal : Integer;
BEGIN
   .........
   Value := MyTotal;     -- copy MyTotal to Value
   Update;               -- use Update to modify Value
   MyTotal := Value;     -- copy updated Value to MyTotal
```

To use the second version to update a variable `MyTotal` all we would need to do is

```
   -- no need for a global variable Value
   MyTotal : Integer;
BEGIN
   .........
   Update(Value => MyTotal);     -- use Update to modify MyTotal
```

In this case the copying in and out between AP `MyTotal` and FP `Value` is done automatically.

Cohesion and Loose Coupling

A section of program code is a candidate for procedurisation if

it performs a single clearly identified action -- **cohesion**

it requires a relatively small amount of information to be transferred between the procedure and its caller and hence relatively few parameters will be required -- **loosely coupled** with its caller.

Hence if a procedure performs two (or more) different tasks consider splitting it into two (or more) simpler procedures.

If a procedure has too many parameters:

consider splitting it into simpler procedures each with fewer parameters

consider altering the placement I/O steps so that data is input or output near to where it is processed. Thus consider:

reducing the number `IN` mode parameters by moving `Get` steps inside the procedure from the caller.

reducing the number of `OUT` mode parameters by moving `Put` steps inside the procedure from the caller.