University of Glasgow

11 Procedural abstraction

- Function procedures
- Proper procedures
- Parameters and arguments

University of Glasgow

Abstraction

- In programming, **abstraction** means the distinction between *what* a program-unit does and *how* it does it.
- This supports a separation of concerns between the implementor (who codes the program-unit) and the application programmer (who uses it).
- **Program-units** include:
  - procedures (here)
  - packages, abstract data types, classes *(see §12)*
  - generic packages and classes *(see §13)*.

University of Glasgow

Proper procedures *vs* function procedures

- A **proper procedure** (or just **procedure**) embodies a *command* to be *executed*.
  - A procedure call is a command.
  - It causes the procedure's body to be executed.
  - Its net effect is to update some variables.
- A **function procedure** (or just **function**) embodies an *expression* to be *evaluated*.
  - A function call is an expression.
  - It causes the function's body to be evaluated.
  - Its net effect is to yield a value (the function's *result*).

University of Glasgow

Procedures in PLs

- Imperative PLs usually support both proper procedures and function procedures. and Fun
  - In Pascal and Ada, proper procedures and function procedures are syntactically distinct.
  - In C and Java, the only distinction is that a proper procedure's result type is VOID.
- Functional PLs support function procedures only.
- OO PLs also support procedures, in the guise of *methods*:
  - Static methods are procedures exported by classes.
  - Instance methods are procedures attached to objects.

University of Glasgow

Example: proper procedure

- Proper procedure in C:

```
void print (Date date)
{   int y = date.y,
        m = date.m,
        d = date.d;
    printf ("%d4-%d2-%d2", y, m, d);
}
```

The procedure body is a block-command.

University of Glasgow

Example: function procedures

- Function in Haskell:

```
power :: (Float, Int) -> Float
power (b, n) =
   if n == 0
   then 1.0
   else b * power(b, n-1)
```

The function's body is an expression.

- Function in C:

```
float power (float b, int n)
{   float p = 1;
    while (n > 0) {
        p *= b; n--;
    }
    return p;
}
```

The function's body is a block-command.

University of Glasgow

Design of function procedures *(1)*

- In most imperative and OO PLs, the function's body is syntactically a *block-command*. This is executed until a **return** determines the function's result.
- Pros and cons:
  - \+ The full expressive power of commands is available to define the function.
  - – This is a roundabout way to compute a result.
  - – A **return** might never be executed.
  - – Side effects are possible.

University of Glasgow

Design of function procedures *(2)*

- In functional PLs, the function's body is syntactically an expression. This is evaluated to yield the function's result.
- Pros and cons of this design:
  - \+ This design is simple and natural.
  - – Expressive power is limited, unless the PL has conditional expressions, iterative expressions, etc.

University of Glasgow

Parameters and arguments *(1)*

- An **argument** is a value (or other entity) that is passed to a procedure.
- An **actual parameter** is an expression that yields an argument.
- A **formal parameter** is an identifier through which a procedure can access an argument.
- What may be passed as arguments?
  - values (in all PLs)
  - variables, or pointers to variables (in many PLs)
  - procedures, or pointers to procedures (in some PLs).

University of Glasgow

Parameters and arguments *(2)*

- A **parameter mechanism** is a means by which a formal parameter provides access to the corresponding argument.
- Different PLs support a bewildering variety of parameter mechanisms: value, result, value-result, constant, variable, procedural, and functional parameters.
- These can all be understood in terms of two underlying concepts:
  - copy parameter mechanisms
  - reference parameter mechanisms.

University of Glasgow

Copy parameter mechanisms *(1)*

- With a **copy parameter mechanism**, a value is *copied* into and/or out of a procedure:
  - The formal parameter *FP* is bound to a local variable of the procedure.
  - A value is copied into that local variable on calling the procedure; or copied out of that local variable (to an argument variable) on return.
- Principal copy parameter mechanisms:
  - copy-in parameter
  - copy-out parameter.

University of Glasgow

Copy parameter mechanisms *(2)*

- **Copy-in parameter** (or value parameter):
  - The argument is a value.
  - On call, a local variable is created and initialized with the argument value.
  - On return, that local variable is destroyed.
- **Copy-out parameter** (or result parameter):
  - The argument is a variable.
  - On call, a local variable is created but not initialized.
  - On return, that local variable's final value is assigned to the argument variable, then the local variable is destroyed.

Example: copy-in parameters in C

- C function:

```
void print (Date date) {
    printf ("%d-%d-%d",
        date.y, date.m,
        date.d);
}
```

Local variable `date` is initialized to the argument value.

- Call:

```
Date today = {2008, 11, 5};
print (today);
```

The argument value is the triple (2008, 11, 5).

University of Glasgow

Reference parameter mechanisms *(1)*

- With a **reference parameter mechanism**, the formal parameter is a *reference* to the argument.
  - The formal parameter *FP* is bound to a reference to the argument.
  - Every access to *FP* is an indirect access to the argument.
- Principal reference parameter mechanisms:
  - constant parameters
  - variable parameters
  - procedural parameters.

Reference parameter mechanisms *(2)*

- **Constant parameter**:
  the argument is a *value*.
- **Variable parameter**:
  the argument is a *variable*.
- **Procedural parameter**:
  the argument is a *procedure*.

Thus any inspection of *FP* is actually an indirect inspection of the argument value.

Thus any access (inspection or update) to *FP* is actually an indirect access to the argument variable.

Thus any call to *FP* is actually an indirect call to the argument procedure.

University of Glasgow

Example: reference parameters in Java

- Java method:

```
void print (Date date) {
    out.print (date.y
        & "-" & date.m
        & "-" & date.d);
    date.y++;
}
```

`date` is a reference to the argument object.

- Call:

```
Date today = new Date (2008, 11, 5);
print (today);
```

The argument is the object to which `today` refers.

University of Glasgow

C parameter mechanisms

- C supports only the copy-in parameter mechanism.
- However, we can achieve the *effect* of a variable parameter by passing a pointer:
  - If a C function has a parameter of type $T*$, the corresponding argument must be a pointer to a variable of type T. The function can then indirectly inspect or update that variable.

University of Glasgow

Java parameter mechanisms

- Java supports the copy-in parameter mechanism for primitive types (such as **`int`**, **`float`**, …).
- *In effect*, Java supports the reference parameter mechanism for object types (such as *`T`*`[]`, `String`, `List`, …).