CENG 242 - Chapter 5: Procedural Abstraction (Abstraction)

Burak Metehan Tunçel - May 2022

Abstraction: Make a program or design reusable by enclosing it in a body, hiding the details, and defining a mechanism to access it. Main idea is separating the usage and implementation of program segments. It is vital in large scale programming. Abstraction is possible in any discipline involving design.

Purpose

- Details are confusing
- Details may contain more error
- Repeating same details increase complexity and errors
- Abstraction philosophy: *Declare once, use many times!*
- *Code reusability* is the ultimate goal
- Parameterization improves power of abstraction

1 Function Procedures and Proper Procedures

A ***procedure*** is an entity that embodies a computation. In particular, a *function procedure* embodies an expression to be evaluated, and a *proper procedure* embodies a command to be executed. The embodied computation is to be performed whenever the procedure is called.

A procedure is often defined by one programmer (the *implementer*) and called by other programmers (the *application programmers*). The application programmers are concerned only with the procedure's *observable behavior*, in other words the outcome of calling the procedure. The implementer is concerned with how that outcome is to be achieved, in other words the choice of algorithm.

Note that the *methods* of object-oriented languages are essentially procedures by another name. The only difference is that every method is associated with a class or with a particular object.

1.1 Function Procedures

A ***function procedure*** (or simply *function*) embodies an expression to be evaluated. When called, the function procedure will yield a value known as its *result.* The application programmer observes only this result, not the steps by which it was computed.

A `C` or `C++` function definition has the form:

$$T\ I\ (FPD_1, \ldots, FPD_n)\ B$$

where

- I is the function's identifier
- FPD_i are *formal parameter* declarations
- T is the result type
- B is a block command called the function's **body**.

B must contain at least one return of the form "**return** E;", where E is an expression of type T. The function procedure will be called by an expression of the form "$I(AP_1, \ldots, AP_n)$;", where the AP_i are *actual parameters.*

In general, a function call can be understood from two different points of view:

- The *application programmer*'s view of the function call is that it will map the arguments to a result of the appropriate type. Only this mapping is of concern to the application programmer.
- The *implementer*'s view of the function call is that it will evaluate or execute the function's body, using the function's formal parameters to access the corresponding arguments, and thus compute the function's result. Only the algorithm encoded in the function's body is the implementer's concern.

1.2 Proper Procedures

A ***proper procedure*** embodies a command to be executed, and when called will update variables. The application programmer observes only these updates, not the steps by which they were effected.

A `C` or `C++` proper procedure definition has the form:

$$\textbf{void}\ I\ (FPD_1, \ldots, FPD_n)\ B$$

where

- I is the function's identifier
- FPD_i are the *formal parameter* declarations
- B is a block command called the procedure's ***body.***

The procedure will be called by a command of the form "$I(AP_1, \ldots, AP_n)$;", where the AP_i are *actual parameters.*

In general, a procedure call can be understood from two different points of view:

- The *application programmer*'s view of the procedure call is its final outcome, which is that certain variables are updated. Only this outcome is of concern to the application programmer.
- The *implementer*'s view of the procedure call is that it will execute the procedure's body, using the procedure's formal parameters to access the corresponding arguments. Only the algorithm encoded in the procedure's body is the implementer's concern.

1.3 The Abstraction Principle

We may summarize the preceding subsections as follows:

- A *function procedure* abstracts over an *expression*. That is to say, a function procedure has a body that is an expression (at least in effect), and a function call is an expression that will yield a result by evaluating the function procedure's body.
- A *proper procedure* abstracts over a command. That is to say, a proper procedure has a body that is a command, and a procedure call is a command that will update variables by executing the proper procedure's body.

Thus there is a clear analogy between function and proper procedures. We can extend this analogy to construct other types of procedures. The ***Abstraction Principle*** states:

> *It is possible to design procedures that abstract over any syntactic category, provided only that the constructs in that syntactic category specify some kind of computation.*

1.3.1 Selector Abstraction

For instance, a variable access refers to a variable. We could imagine designing a new type of procedure that abstracts over variable accesses. Such a procedure, when called, would yield a variable. In fact, such procedures do exist, and are called ***selector procedures***:

- A *selector procedure* abstracts over a *variable access*. That is to say, a selector procedure has a body that is a variable access (in effect), and a selector call is a variable access that will yield a variable by evaluating the selector procedure's body.

Selector procedures are uncommon in programming languages, but they are supported by `C++`.

As an example, [..] is a operator that selects elements of an array. There can be user defined selectors on user defined structures.

```
struct List {
  int data;
  List *next;

  int &operator[](int el) {
    int i; List *p = this;
    for (i = 1 ; i < el ; i++)
      p = p->next;  /* take the next element */

    return p->data;
  };
  ...
};

List h;
...
h[1] = h[2] + 1;
```

Code 1

In Python, `__setitem__(k,v)` implements l-value, `__getitem__(k,v)` r-value selector.

```
class BSTree:
  def __init__(self):
    self.node = None
  def __getitem__(self, key):
    if self.node == None:
        raise KeyError
    elif key < self.node[0]:
      return self.left[key]
    elif key > self.node[0]:
      return self.right[key]
    else:
      return self.node[1]
  def __setitem__(self, key, val):
    if self.node == None:
        self.node = (key,val)
        self.left = BSTree()    # empty tree
        self.right = BSTree()   # empty tree
    elif key < self.node[0]:
      self.left[key] = val
    elif key > self.node[0]:
      self.right[key] = val
    else:
      self.node = (key,val)

a = BSTree()
a["hello"] = 4
a["world"] = a["hello"] + 5
```

Code 2

1.3.2 Generic Abstraction

Another direction in which we might push the Abstraction Principle is to consider whether we can abstract over *declarations*. This is a much more radical idea, but it has been adopted in some modern languages such as `C++` and `JAVA`. We get a construct called a ***generic unit***:

- A *generic unit* abstracts over a *declaration*. That is to say, a generic unit has a body that is a declaration, and a generic instantiation is a declaration that will produce bindings by elaborating the generic unit's body.

In other words, *same declaration pattern applied to different data types.* A function or class declaration can be adapted to different types or values by using type or value parameters.

```
template <class T> class List {
        T content;
        List *next;
  public: List() { next = NULL };
        void add(T el) { ... };
        T get(int n) { ...};
};

template <class U>
  void swap(U &a, U &b)
    { U tmp; tmp=a; a=b; b=tmp; }
...
List<int> a; List<double> b; List<Person> c;
int t,x; double v,y; Person z,w;
swap(t,x); swap(v,y); swap(z,w);
```

Code 3

1.3.3 Iterator Abstraction

Iteration over a user defined data structure. Python generator example:

```
class BSTree(object):
  def __init__(self):
    self.val = ()
  def inorder(self):
    if self.val == ():
      return
    else:
      for i in self.left.inorder():
        yield i
      yield self.val
      for i in self.right.inorder():
        yield i

v = BSTree()
...
for v in v.inorder():
  print v
```

Code 4

C++ Iterators `C++` Standard Template Library containers support *iterators* `begin()` and `end()` methods return iterators to start and end of the data structure. Iterators can be dereferenced as `*iter` or `iter->member`. '`++`' operation on an iterator skips to the next value.

```
for (ittype it = a.begin(); it != a.end(); ++it) {
  // use *it or it->member it->method() in body
}
```

`C++11` added:

```
for (valtype & i : a ) {
  // use directly i as l-value or r-value.
}
```

This syntax is equivalent to:

```
for (ittype it = a.begin() ; it != a.end(); it++) {
  valtype & i = *it;
  // use directly i as l-value or r-value
}
```

Following is the general examples:

```
template<class T>
class List {
  struct Node { T val; Node *next;} *list;
public:
  List() { list = nullptr;}
  void insert(const T& v) {
    Node *newnode = new Node;
    newnode->next = list; newnode->val = v;
    list = newnode;
  }
  class Iterator {
    Node *pos;
  public:
    Iterator(Node *p) { pos = p; }
    T & operator*() { return pos->val; }
    void operator++() { pos = pos->next; }
    bool operator!=(const Iterator &it) {
      return pos != it.pos;
    }
  };

  Iterator begin()
    { Iterator it = Iterator(list); return it; }
  Iterator end()
    { Iterator it = Iterator(nullptr); return it; }
};
List<int> a;
// C++11 syntax below
for (int & i : a )
  i *= 2; cout << i << '\n';
```

Code 5

1.4 Abstraction Principle

If any programming language entity involves computation, it is possible to define an abstraction over it

Entity	→	**Abstraction**
Expression	→	Function
Command	→	Procedure
Selector	→	Selector function
Declaration	→	Generic
Command Block	→	Iterator

2 Parameters and Arguments

An ***argument*** is a value or other entity that is passed to a procedure. An ***actual parameter*** is an expression (or other construct) that yields an argument. A ***formal parameter*** is an identifier through which a procedure can access an argument.

In all programming languages, *first-class values can be passed as arguments.* In most languages, *either variables or pointers to variables can be passed as arguments.* In some languages, *either procedures or pointers to procedures can be passed as arguments.*

When a procedure is called, each formal parameter will become associated, in some sense, with the corresponding argument. The nature of this association is called a ***parameter mechanism***. Parameter mechanisms can be understood in terms of two basic concepts:

- A *copy parameter* mechanism binds the formal parameter to a local variable that contains a copy of the argument.
- A *reference parameter* mechanism binds the formal parameter directly to the argument itself.

2.1 Copy Parameter Mechanisms

A ***copy parameter mechanism*** allows for a value to be copied into and/or out of a procedure. The formal parameter FP denotes a local variable of the procedure. A value is copied into FP on calling the procedure, *and/or* is copied out of FP (to an argument variable) on return. The local variable is created on calling the procedure, and destroyed on return.

There are three possible copy parameter mechanisms:

- A ***copy-in parameter*** (also known as a *value parameter*) works as follows. When the procedure is called, a local variable is created and initialized with the argument value. Inside the procedure, that local variable may be inspected and even updated. (However, any updating of the local variable has no effect outside the procedure.)
- A ***copy-out parameter*** (also known as a *result parameter*) is a mirror-image of a copy-in parameter. In this case the argument must be a variable. When the procedure is called, a local variable is created but not initialized. When the procedure returns, that local variable's final value is assigned to the argument variable.
- A ***copy-in-copy-out parameter*** (also known as a *value-result parameter*) combines copy-in and copy-out parameters. In this case also, the argument must be a variable. When the procedure is called, a local variable is created and initialized with the argument variable's current value. When the procedure returns, that local variable's final value is assigned back to the argument variable.

`C`, `C++`, and `JAVA` support *only* copy-in parameters.

2.2 Reference Parameter Mechanisms

A ***reference parameter mechanism*** allows for the formal parameter FP to be bound directly to the argument itself.

Reference parameter mechanisms appear under several guises in programming languages:

- In the case of a ***constant parameter***, the argument must be a value. FP is bound to the argument value during the procedure's activation. Thus any inspection of FP is actually an indirect inspection of the argument value.
- In the case of a ***variable parameter***, the argument must be a variable. FP is bound to the argument variable during the procedure's activation. Thus any inspection (or updating) of FP is actually an indirect inspection (or updating) of the argument variable.
- In the case of a ***procedural parameter***, the argument must be a procedure. FP is bound to the argument procedure during the called procedure's activation. Thus any call to FP is actually an indirect call to the argument procedure.

`C++` does support variable parameters directly. If the type of a formal parameter is $T\&$ (reference to T), the corresponding argument must be a variable of type T.

The choice between reference and copy parameter mechanisms is an important decision for the language designer. Reference parameters have simpler semantics, and are suitable for all types of value (including procedures, which in most programming languages cannot be copied). Reference parameters rely on indirect access to argument data, so *copy parameters are more efficient for primitive types, while reference parameters are usually more efficient for composite types.* (In a distributed system, however, the procedure might be running on a processor remote from the argument data, in which case it may be more efficient to copy the data and then access it locally.)

A disadvantage of variable parameters is that aliasing becomes a hazard. ***Aliasing*** occurs when two or more identifiers are simultaneously bound to the same variable (or one identifier is bound to a composite variable and a second identifier to one of its components). Aliasing tends to make programs harder to understand and harder to reason about.

2.3 The Correspondence Principle

You might have noticed a correspondence between certain parameter mechanisms and certain forms of declaration. For example:

- A *constant parameter* corresponds to a *constant definition*. In each case, an identifier is bound to a first-class value.
- A *variable parameter* corresponds to a *variable renaming definition*. In each case, an identifier is bound to an existing variable.
- A *copy-in parameter* corresponds to an *(initialized) variable declaration*. In each case, a new variable is created and initialized, and an identifier is bound to that variable.

The ***Correspondence Principle*** states:

> *For each form of declaration there exists a corresponding parameter mechanism.*

Note that the converse is not always true.

`C`:

```
int a = p;        ↔ void f(int a) {...}
const int a = p;  ↔ void f(const int a) {...}
```

`Pascal`:

```
var a:  integer; ↔ procedure f(a:integer)
                 ↔ begin
const a:  5;     ↔ ???
???              ↔ procedure f(var a:integer)
                 ↔ begin
```

`C++`:

```
int a = p;        ↔ void f(int a) {...}
const int a = p;  ↔ void f(const int a) {...}
&a = p            ↔ void f(int &a) {...}
```

3 Parameters (*from slides*)

Declaration part:

`abstraction_name` $(FP_1, FP_2, \ldots, FP_n)$

Use part:

`abstraction_name` $(AP_1, AP_2, \ldots, AP_n)$

Formal parameters: Identifiers or constructors of identifiers (patterns in functional languages).

Actual parameters: Expression or identifier based on the type of the abstraction and parameter.

3.1 Parameter passing mechanisms

Programming language may support one or more mechanisms. 3 basic methods:

1. Copy mechanisms (assignment based)
2. Binding mechanisms
3. Pass by name (substitution based)

3.1.1 Copy Mechanisms

Function and procedure abstractions, assignment between actual and formal parameter:

1. **Copy In:**
   On function call: $FP_i \leftarrow AP_i$
2. **Copy Out:**
   On function return: $AP_i \leftarrow FP_i$
3. **Copy In-Out:**
   On function call: $FP_i \leftarrow AP_i$, and
   On function return: $AP_i \leftarrow FP_i$

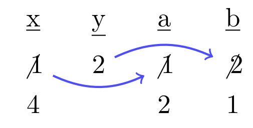
`C` only allows copy-in mechanism. This mechanism is also called as **Pass by value**.

```
int x=1, y=2;
void f(int a, int b) {
  x += a+b;
  a++;
  b=a/2;
}

int main() {
  f(x,y);
  printf("x:%d, y:%d\n",x,y);
  return 0;
}
```

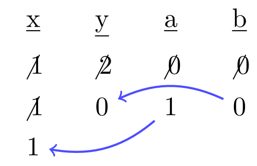
Code 6

Copy In:



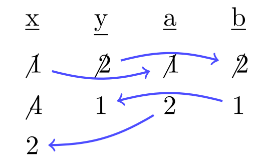
```
x:4, y:2
```

Copy Out:



```
x:1, y:0
```

Copy In-Out:



```
x:2, y:1
```

3.1.2 Binding Mechanisms

Based on binding of the formal parameter variable/identifier to actual parameter value/identifier. Only one entity (value, variable, type) exists with more than one names.

1. **Constant binding:** Formal parameter is constant during the function. The value is bound to actual parameter expression value.
   Functional languages including Haskell uses this mechanism.
2. **Variable binding:** Formal parameter variable is bound to the actual parameter variable. Same memory area is shared by two variable references.
   Also known as **pass by reference**.

The other type and entities (function, type, etc) are passed with similar mechanisms.

```c
int x = 1, y = 2;
void f(int a, int b) {
  x += a + b;
  a++;
  b = a / 2;
}

int main() {
  f(x, y);
  printf("x:%d , y:%d\n",x,y);
  return 0;
}
```

Code 7

Variable binding:

f():a	f():b
x	y
$\cancel{1}$	$\cancel{2}$
$\cancel{4}$	2
5	

```
x:  5, y:2
```

3.1.3 Pass by Name

Actual parameter syntax replaces each occurence of the formal parameter in the function body, then the function body evaluated. `C` macros works with a similar mechanism (by pre-processor).

Mostly useful in theoretical analysis of PL's. Also known as **Normal order evaluation**.

Example (Haskell-like)

```haskell
f x y = if (x < 12) then x*x + y*y + x
        else x + x*x
```

Code 8

Evaluation:

$$
\begin{aligned}
&\texttt{f}\ (3*12+7)\ (24+16*3) \\
&\mapsto \texttt{if}\ ((3*12+7)<12)\ \texttt{then}\ (3*12+7)*(3*12+7)+(24+16*3)*(24+16*3)+(3*12+7) \\
&\qquad \texttt{else}\ (3*12+7)+(3*12+7)*(3*12+7) \\
&\overset{*}{\mapsto} \texttt{if}\ (43<12)\ \texttt{then}\ \ldots \\
&\mapsto \texttt{if}\ (false)\ \texttt{then}\ \ldots \\
&\mapsto (3*12+7)+(3*12+7)*(3*12+7) \\
&\overset{*}{\mapsto} (3*12+7)+43*(3*12+7) \\
&\mapsto \ldots \mapsto 1892 \quad \texttt{12 steps}
\end{aligned}
$$

Normal order evaluation is mathematically natural order of evaluation.

Most of the PL's apply ***eager evaluation***: Actual parameters are evaluated first, then passed.

$$\begin{aligned}
&\texttt{f}\ \ (3*12+7)\ (24+16*3)\\
&\mapsto \texttt{f}\ \ (36+7)\ (24+16*3)\\
&\overset{*}{\mapsto} \texttt{f}\ \ 43\ 72\\
&\mapsto \texttt{if}\ \ (43<12)\ \texttt{then}\ \ 43*43+72*72+43\\
&\quad\ \texttt{else}\ \ 43+43*43\\
&\mapsto \texttt{if}\ \ (false)\ \texttt{then}\ \ ...\\
&\mapsto 43+43*43 \overset{*}{\mapsto} 1892 \quad \texttt{8 steps}
\end{aligned}$$

Church–Rosser Property:

> If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing eager and normal-order evaluation), then all of these evaluation orders yield the same result.

In λ-calculus, all orders reduce the same normal form.

$$\begin{tikzcd}
& & & \lambda x.(\lambda y.y+(\lambda x.x+1\ y)\ \ (x+2))\ \ 5 \arrow[dll, |->] \arrow[dd, |->] \arrow[drr, |->] & & & \\
& \lambda y.y+(\lambda x.x+1\ \ y)\ \ (5+2) \arrow[dl, |->] \arrow[dddl, |->, bend left] & & & & \lambda x.(\lambda y.y+y+1\ \ (x+2))\ \ 5 \arrow[ddd, |->] \arrow[ddr, |->] & \\
5+2+(\lambda x.x+1\ \ (5+2)) \arrow[dddrrr, |->] & & & \lambda x.x+2+(\lambda x.x+1\ \ (x+2))\ 5 \arrow[dl, |->] \arrow[dr, |->] & & & \\
& & 5+2+(\lambda x.x+1\ \ (5+2)) \arrow[ddr, |->] & & \lambda x.x+2+x+2+1\ \ 5 \arrow[ddl, |->] & & \lambda y.y+y+1\ \ (5+2) \arrow[ddlll, |->, bend left] \\
\lambda y.y+y+1\ \ (5+2) \arrow[drrr, |->] & & & & & \lambda x.x+2+x+2+1\ \ 5 \arrow[dll, |->] & \\
& & & 5+2+5+2+1 & & &
\end{tikzcd}$$

Haskell implements **Lazy Evaluation** order.

- Eager evaluation is faster than normal order evaluation but violates Church-Rosser Property.
- Lazy evaluation is as fast as eager evaluation but computes same results with normal order evaluation (unless there is a side effect).

Lazy evaluation expands the expression as normal order evaluation however once it evaluates the formal parameter value other evaluations use previously found value:

$$\begin{aligned}
&\texttt{f}\ \ (3*12+7)\ (24+16*3)\\
&\mapsto \texttt{if}\ \ (x:(3*12+7)<12)\ \texttt{then}\ \ x:(3*12+7)*x:(3*12+7)+y:(24+16*3)*y:(24+16*3)+x:(3*12+7)\\
&\quad\ \texttt{else}\ \ x:(3*12+7)+x:(3*12+7)*x:(3*12+7)\\
&\overset{*}{\mapsto} \texttt{if}\ \ (x:43<12)\ \texttt{then}\ \ x:43*x:43+y:(24+16*3)*y:(24+16*3)+x:43\\
&\quad\ \texttt{else}\ \ x:43+x:43*x:43\\
&\mapsto \texttt{if}\ \ (false)\ \texttt{then}\ \ ...\\
&\mapsto x:43+x:43*x:43\\
&\mapsto x:43+1849 \mapsto 1892 \quad \texttt{7 steps}
\end{aligned}$$

In lazy evaluation, parameters are passed by name but compiler keeps evaluation state of them. Parameter value is store once it is evaluated. Further evaluations use that.v

Python implementation. First delay evaluation of expressions. Convert to functions:

exp → **lambda** : exp
η expansion. Function version is also called **thunk**.

Inside function, call these functions to evaluate the expression v

```
def E(thunk):
  if not hasattr(thunk,"stored"):
    thunk.stored = thunk()      # evaluate and store
  return thunk.stored           # use stored value
def f(x,y):
  if E(x) < 10:                # call E() on all evaluations
    return E(x)*E(x)+E(y)
  else:
    return E(x)*E(x)+E(x)
f(lambda : 3*32+4, lambda: 4/0)     # call by converting to function
```

Code 9

Delayed evaluation in normal order or lazy evaluation enables working on infinite values.