

Programming Language Concepts

Abstraction

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

1 Abstraction

- Function and Procedure Abstractions
- Selector Abstraction
- Generic Abstraction
- Iterator Abstraction
- Iterator Abstraction

2 Abstraction Principle

3 Parameters

4 Parameter Passing

Mechanisms

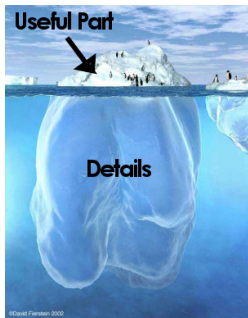
- Copy Mechanisms
- Binding Mechanisms
- Pass by Name

5 Evaluation Order

6 Infinite Values

7 Correspondence Principle

Abstraction



- Iceberg: Details at the bottom, useful part at the top of the ocean. Animals do not care about the bottom.
- User: "how do I use it?", Developer: "How do I make it work?"
- User: "what does it do?", Developer: "How does it do that?"
- **Abstraction:** Make a program or design reusable by enclosing it in a body, hiding the details, and defining a mechanism to access it.
- Separating the usage and implementation of program segments.
- Vital large scale programming.

- Abstraction is possible in any discipline involving design:
- radio tuner. Adjustment knob on a radio is an abstraction over the tuner element, frequency selection.
- An ATM is an abstraction over complicated set of bank transaction operations.
- Programming languages can be considered as abstraction over machine language.
- ...

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

Function and procedure abstractions

- The computation of an expression is the detail (algorithm, variables, etc.)
- Function call is the usage of the detail
- Functions are **abstractions over expressions**
- void functions of C or procedure declarations of some languages
- No value but contains executable statements as detail.
- Procedures are **abstractions over commands**
- Other type of abstractions possible?

Selector abstraction

- arrays: `int a[10][20]; a[i]=a[i]+1;`
- `[..]` operator selects elements of an array.
- User defined selectors on user defined structures?
- Example: Selector on a linked list:

```
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;
```

- C++ allows overloading of `[]` operator for classes.

- Python `__setitem__` (k,v) implements l-value, `__getitem__` (k) 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
```



```

class BST {
    struct Node { string key; double val;
                  Node *left, *right;} *node;
public:
    BST() { node = NULL;};
    double & operator[](const string &k) {
        Node **parent = NULL, *p = node, *newnode;
        while (p != NULL) {
            if (k < p->key) {
                parent = &p->left; p = p->left;
            } else if (k > p->key) {
                parent = &p->right; p = p->right;
            } else return p->val;
        }
        newnode = new Node;
        newnode->left = newnode->right = NULL;
        newnode->key = k;
        if (parent == NULL) node = newnode;
        else *parent = newnode;
        return newnode->val;
    }
};

BST a;
a["carrot"] = 3; a["onion"] = 4;
a["patato"] = a["onion"] + 2;

```

Generic abstraction

- Same declaration pattern applied to different data types.
- **Abstraction over declaration.** 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);
```

Iterator abstraction

- Iteration over a user defined data structure. **Ruby** example:

```
class Tree
  def initialize(v)
    @value = v ; @left = nil ; @right = nil
  end
  def traverse
    @left.traverse {|v| yield v} if @left != nil
    yield @value           # block argument replaces
    @right.traverse {|v| yield v} if @right != nil
  end
end

a=Tree.new(3) ; l=[]
a.traverse { |node|      # yield param
               print node # yield body
               l << node  # yield body
            }
```

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
```

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

# C++ iterators

```
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'; }
for (const char * s : { "ankara","istanbul","izmir" }) {
 cout << s ; }
```

# 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          |

# Parameters

- Many purpose and behaviors in order to take advantage of “declare once use many times”.
- **Declaration part:** `abstraction_name(Fp1, Fp2, ..., Fpn)`  
**Use part:** `abstraction_name(Ap1, Ap2, ..., Apn)`
- 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
- **Question:** How actual and formal parameters relate/communicate?
- Programming language design should answer
- **Parameter passing mechanisms**



# 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)

# 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;
}

```

Copy In:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 1        | 2        |
| 4        |          | 2        | 1        |

x:4, y:2

Copy Out:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 0        | 0        |
| 1        | 0        | 1        | 0        |
| 1        |          |          |          |

x:1, y:0

Copy In-Out:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 1        | 2        |
| 4        | 1        | 2        | 1        |
| 2        |          |          |          |

x:2, y:1

# 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.

```

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;
}

```

Variable binding:

| f():a /   | f():b / |
|-----------|---------|
| x         | y       |
| <hr/>     | <hr/>   |
| 1         | 2       |
| 4         | 2       |
| 5         |         |
| x: 5, y:2 |         |

# Pass by name

- Actual parameter syntax replaces each occurrence 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)

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

Evaluation:  $f\ (3*12+7)\ (24+16*3) \mapsto \text{if } ((3*12+7)<12) \text{ then } (3*12+7)*(3*12+7)+(24+16*3)*(24+16*3)+(3*12+7) \text{ else } (3*12+7)+(3*12+7)*(3*12+7) \xrightarrow{*} \text{if } (43<12) \text{ then } \dots \mapsto \text{if } (\text{false}) \text{ then } \dots \mapsto (3*12+7)+(3*12+7)*(3*12+7) \xrightarrow{*} (3*12+7)+43*(3*12+7) \mapsto \dots \mapsto 1892$  (12 steps)

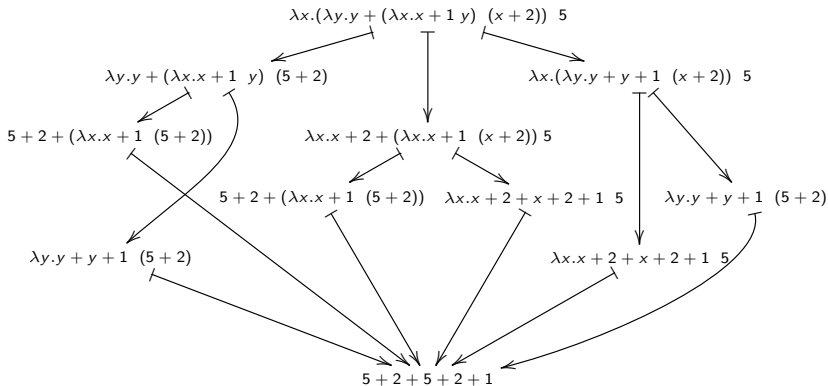
# Evaluation Order

- **Normal order evaluation** is mathematically natural order of evaluation.
- Most of the PL's apply **eager evaluation**: Actual parameters are evaluated first, then passed.

$f(3*12+7)(24+16*3) \mapsto f(36+7)(24+16*3) \xrightarrow{*} f\ 43\ 72 \mapsto \text{if } (43 < 12)$   
 $\text{then } 43*43+72*72+43 \text{ else } 43+43*43 \mapsto \text{if } (\text{false}) \text{ then } \dots \mapsto$   
 $43+43*43 \xrightarrow{*} 1892 \quad (8 \text{ steps})$

- Consider “ $g \ x \ y = \text{if } x > 10 \text{ then } y \text{ else } x$ ” for  $g \ 2 \ (4/0)$
- Side effects are repeated in NOE.
- **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  $\lambda$ -calculus, all orders reduce the same normal form.





- 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:

```
f (3*12+7) (24+16*3) \mapsto if (x:(3*12+7)<12) then
x:(3*12+7)*x:(3*12+7)+y:(24+16*3)*y:(24+16*3)+x:(3*12+7) else
x:(3*12+7)+x:(3*12+7)*x:(3*12+7) $\xrightarrow{*}$ if (x:43<12) then
x:43*x:43+y:(24+16*3)*y:(24+16*3)+x:43 else x:43+x:43*x:43 \mapsto if
(false) then ... \mapsto x:43+x:43*x:43 \mapsto x:43+1849 \mapsto 1892 (7 steps)
```

# 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.
- Python implementation. First delay evaluation of expressions. Convert to functions:  
 $exp \rightarrow \text{lambda} : exp$   
 $\eta$  expansion. Function version is also called **thunk**.
- Inside function, call these functions to evaluate the expression.

```
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
```

# Infinite Values

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

```
take _ [] = []
take n (a:r) | n == 0 = []
 | otherwise = a : take (n-1) r
x = (1:2:x)
```

```
take 3 x \mapsto take 3 (1:2:x) \mapsto 1:take (3-1) (2:x) \mapsto
1:2:take (2-1) x \mapsto 1:2:take 1 (1:2:x) \mapsto 1:2:1:take (1-1) (2:x) \mapsto
1:2:1:[]
```

- Programmers can take advantage of this. Construct an infinitely value, take as many as program needs. For example expand  $\pi$  in an infinite value, stop when desired resolution achieved.

# Correspondence Principle

## ■ Correspondence Principle:

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

## ■ C:

|                             |                   |                                    |
|-----------------------------|-------------------|------------------------------------|
| <code>int a=p;</code>       | $\leftrightarrow$ | <code>void f(int a) {</code>       |
| <code>const int a=p;</code> | $\leftrightarrow$ | <code>void f(const int a) {</code> |

## ■ Pascal:

|                              |                   |                                               |
|------------------------------|-------------------|-----------------------------------------------|
| <code>var a: integer;</code> | $\leftrightarrow$ | <code>procedure f(a:integer) begin</code>     |
| <code>const a:5;</code>      | $\leftrightarrow$ | <code>??? {</code>                            |
| <code>???</code>             | $\leftrightarrow$ | <code>procedure f(var a:integer) begin</code> |

## ■ C++:

|                             |                   |                                    |
|-----------------------------|-------------------|------------------------------------|
| <code>int a=p;</code>       | $\leftrightarrow$ | <code>void f(int a) {</code>       |
| <code>const int a=p;</code> | $\leftrightarrow$ | <code>void f(const int a) {</code> |
| <code>int &amp;a=p;</code>  | $\leftrightarrow$ | <code>void f(int &amp;a) {</code>  |