TDDA69 Data and Program Structure
Imperative Programming Techniques

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

Lecture content

- Programming with states: imperatively
- Data abstraction
- Modular Programming
- Object Oriented Programming
  - Object-Oriented Concepts
- Memory and threading

Programming with states: imperatively

Imperative vs declarative

- Declarative
  - No state
  - Expresses logic of computation without control flow
- Imperative
  - States
  - Describes computation in term of statements that change the internal state

Imperative syntax construction

- *A state is a **sequence** of values in time that contains the intermediate results of a desired computation*.
- Would not it make sense to express computation also as a **sequence**?
- Syntax abstractions to manipulate state: loop, if

Loops

- With a loop:

```
function factorial(n)
{
  var r = 1;
  for(var i = 2; i <= n; ++i)
  {
    r *= i;
  }
  return r;
}
```

- With a recursive call:

```
function factorial(n)
{
  return (n===0) ? 1 : n * factorial(n-1)
}
```


Side-effects

- Generally no-pure functions in imperative programming, but:

```
function factorial(n)
{
  var r = 1;
  for(var i = 2; i <= n; ++i)
  {
    r *= i;
  }
  return r;
}
```

- No side-effect, always return the same result for the same arguments

Random function

```
function make_rand(seed)
-> rand
{
  cell X = seed;
  define A = 333667;
  define B = 213453321;
  define RAND_MAX =
1000000;
  rand = function() ->
ret
  {
    ret = (X = A*X+B) %
RAND_MAX;
  }
}
```

```
thread make_rand(seed) -> ret
{
  define s = stream();
  ret = s.output();
  define A = 333667;
  define B = 213453321;
  define RAND_MAX = 1000000;
  function p(X)
  {
    s.send(X = A*X+B) % M;
  }
  p(seed);
}
```


Reasoning with state

- In the general case, we cannot ignore side-effects, how can we reason?
- Simple with simple programs, impossible with larger ones
- What if a state is visible through the entire program?

```
cell a = 0;
function () -> y { y = a; }
a = 2;
```


Reasoning with state: invariant

- An invariant is a condition that is assumed to hold at a specific point in the program
- Invariants are relationships among the variables of a program that always hold.

```
cell a = 0;
function () -> y { assert(a < 2); y
= a; }
a = 2;
```

- Avoid unwanted side effects!

Loop invariants

- A *loop invariant* is a relationship among the variables in a loop that holds at the begining and at the end of every iteration.

```
for(var i = 0; i < 10; ++i)
{
  assert(i >= 0 && i < 10);
}
```

Proving properties in imperative programming

- Also used for proving:
- ```
  function power(b, n) {
    int result = 1;
    for(int i = 0; i < n; ++i)
    {
      result *= b;
    }
    return result;
  }
  ```
- Devise a *loop invariant*:
  - $(n \geq i) \wedge (result = b^i)$
  - Prove that it is true for the first loop iteration
  - Prove that each loop iteration preserves it
    Assume that $(n \geq i) \wedge (result = b^i)$
    Prove that $(n \geq j) \wedge (result = b^j)$ with $j = i + 1$

Data structure invariant

- Data structure invariants are the invariant relations among the fields of any object of a particular class
- List:
  - operations for creating, empty, append, prepend: no invariant
  - operations for head of list, pop: invariant is that the list is non empty

Encapsulation

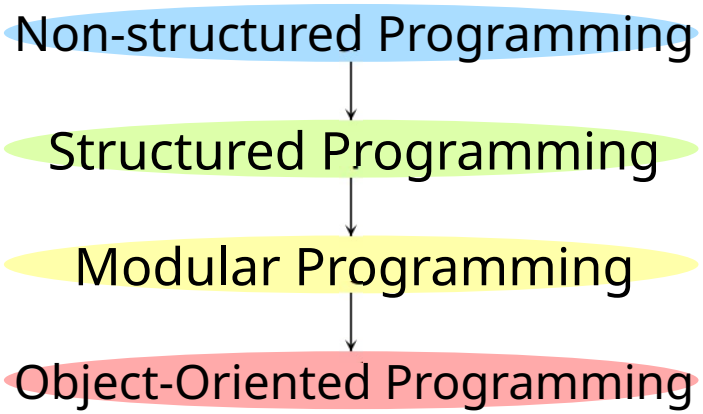
- How to get protection from side-effects
- In the rand function, only rand can update the state
- It is called *encapsulation*!

Data abstraction
  ```

The goals of data abstraction

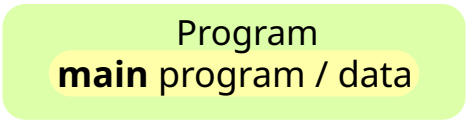
- Encapsulation
  - Hide internals from the interface
- Compositionality
  - Combine parts to make new parts
- Instantiation/invocation
  - Create new instances of parts

Program Organizing Techniques



Non-structured Programming

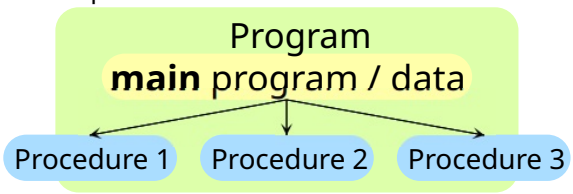
- The **main** program directly operates on global data



- Impractical when the program gets sufficiently large
- The same statement sequence must be copied if it is needed several times

Procedural Programming

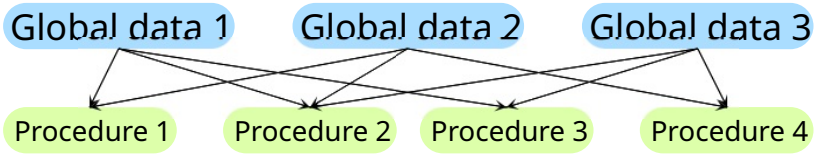
- Combine a sequence of statements into a procedure with calls and returns
- The main program coordinates calls to procedures and hands over appropriate data as parameters



- Tasks are a collection of states, structures and procedures

Limitations of Structured Programming

- Unrestricted access to global data

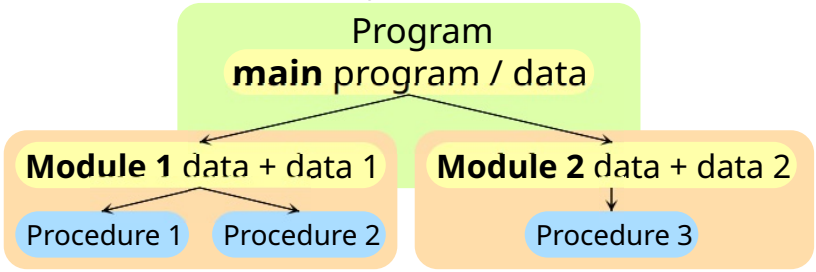


- Attributes and behaviors are seperated
- Not extensible

Modular Programming

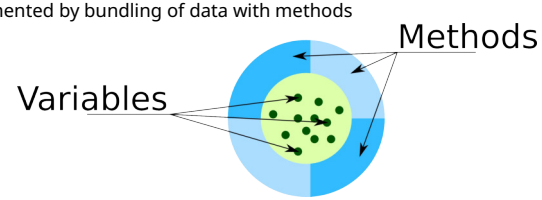
Modular Programming

- Procedures of a common functionality are grouped together into separate modules.
- The main program coordinates calls to procedures in separate modules and hands over data as parameters.



Encapsulation

- Hide internals from the interface
  - Can be implemented by bundling of data with methods



  - It provides abstraction
- Two common levels of hiding:
  - Private: only accessible to the object
  - Public: accessible to the world
- How to implement abstraction in KL?
  - +state+closure

Linked-list

```
function new_list()
{
  return { cell head = null; cell tail = null; }
}
function list_push(list, _value)
{
  cell node = { cell value = _value; cell next =
null };
  if(list.tail)
  {
    list.tail.next = node;
    list.tail = node;
  } else {
    list.head = node;
    list.tail = node;
  }
}
...
```

- No encapsulation

Bundling data with method

```
function new_list()
{
  return { cell head = null; cell tail = null;
    function push(_value)
    {
      cell node = { cell value = _value; cell
next = null };
      if(list.tail)
      {
        tail.next = node;
        tail = node;
      } else {
        head = node;
        tail = node;
      }
    }
    ...
  };
}
```

- No encapsulation

Private data, attempt

```
function new_list()
{
  cell priv = { cell head = null; cell tail =
null; }
  return {
    function push(_value)
    {
      cell node = { cell _value; cell next = null
};
      if(list.tail)
      {
        priv.tail.next = node;
        priv.tail = node;
      } else {
        priv.head = node;
        priv.tail = node;
      }
    }
    ...
  };
}
```

- No encapsulation:

```
cell list =
new_list();
list.priv.head
= 'breaking
the data
structure'
```


Encapsulation in C

- Using int_list_t*:

```
struct int_list_t;
int_list_t* new_int_list();
void int_list_push(int_list_t*,
int _value);
...
```

- Internaly:

```
typedef struct
{
  int value;
  int_list_node* next;
} int_list_node;
typedef struct
{
  int_list_node* head;
  int_list_node* tail;
} int_list;
```

```
int_list_t* new_int_list()
{
  int_list* l =
malloc(sizeof(int_list));
  l->head = NULL;
  l->tail = NULL;
  return l;
}
void int_list_push(int_list_t*
_l, int _value)
{
  ...
}
```

Private data, attempt

```
function new_list()
{
  cell priv = { cell head = null; cell tail =
null; };
  cell self = {
    function push(_value)
    {
      cell node = { cell value = _value, cell
next = null };
      if(list.tail)
      {
        priv.tail.next = node;
        priv.tail = node;
      } else {
        priv.head = node;
        priv.tail = node;
      }
    }
    ...
  };
  clear(priv);
  return self;
}
```

- Does not work:

```
cell list =
new_list();
list.push(1); //
triggers an
error that priv
is not defined
```


Private data, attempt #3

```
function new_list()
{
  cell priv = { cell head = null; cell tail = null;
    function push(value)
    {
      cell node = { cell value; cell next = null };
      if(list.tail)
      {
        tail.next = node;
        tail = node;
      } else {
        head = node;
        tail = node;
      }
    }
    ...
    };
  cell self = {
    push = priv.push
    ...
  };
  clear(priv);
  return self;
}
```


Private data, alternative

```
function new_list()
{
  cell self = {};
  (function() {
    cell head = null;
    cell tail = null;
    define self.push = function(value)
    {
      cell node = { value, next = null };
      if(list.tail)
      {
        tail.next = node;
        tail = node;
      } else {
        head = node;
        tail = node;
      }
    }
    ...
  })();
  return self;
}
```


Object Oriented Programming

Object-Oriented Concepts

What is an Object?

- Real-world objects are composed of attributes and behaviors
  - Attributes: color, size,
  - Behaviors: accelerate, brake,
- Program objects
  - Attributes are states/
  - Behaviors are

Object-Oriented Concepts

- Class
  - Instantiation/invocation
- Encapsulation
  - Compositionality
- Polymorphism
- Inheritance

Class

- An object is a specific *instance* (ie this car, this computer...)
- A class is a type of *object* (ie a car, a computer...)
  - It is a blueprint of object
- The class defines:
  - The default set of variables
  - The default set of functions
  - An initialisation function (called *constructor*
- The object is an *instance* of a class
  - It points to a specific memory location with the actual values for variables

Class Car

- class Car:
  - attributes:
    color can be red, green, blue...
    speed from 0 to max_speed
    max_speed and max_acceleration are constant
  - functions:
    accelerate increase *speed* with *max_acceleration*
    brake decrease *speed* with *max_acceleration*
- object:
  - red_audi := { color:'red', max_speed: 200, max_acceleration: 10 }
  - blue_volvo := { color:'blue', max_speed: 180, max_acceleration: 7 }

Inheritance

- Some classes have common part
  - Audi is a car, Volvo is a car, but Volvo and Audi are two different type of cars
- Inheritance allows to define a class in term of an other *super*-class
- Sub-classes
  - can add new variables and functions
  - override variables and functions from the super-class

Inheritance from the class Car

- class Car: *color*, *speed*, *max_speed* and *max_acceleration*
- class Audi: class Car with *max_speed := 200* and *max_acceleration := 10*
- class Volvo: class Car with *max_speed := 180* and *max_acceleration := 7*

Encapsulation

- Encapsulation with inheritance
- Three common levels of hiding:
  - Private: only accessible to the object
  - Protected: only accessible to the object and through inheritance
  - Public: accessible to the world

Car example

```
function Car(self, protected, _color, max_speed, max_acceleration)
{
  cell protected.color = _color;
  cell priv = {
    cell speed = 0.0;
    function accelerate()
    {
      speed = Math.min(max_speed, speed + max_acceleration)
    }
    function deccelerate()
    {
      speed = Math.max(0, speed - max_acceleration)
    }
  };
  define self.accelerate = priv.accelerate;
  define self.deccelerate = priv.deccelerate;
  clear(priv);
};
 var obj = {};
 Car(obj, {}, 'red', 200, 10)
```


Convenient object creation

- A new operator:

```
function new(constructor, arguments...)
{
  cell self = {}
  constructor(self, {}, arguments...);
  return self;
}
new(Car, 'red', 200, 10)
```


Inheritence

- For inheritance, all we need is to call the super class constructor

```
function Audi(self, protected, color)
{
  Car(self, protected, color, 200, 10)
})
new(Audi, 'blue');
```


Polymorphism

- *polymorphism* is the ability to appear in many forms
- It is the ability to call a function on an *data structure* that will execute different code for different data structure
- Define generic interface
  - Example:
    List push, pop
    Stack push, pop
- It is one way to define generic and composable code
- With KL, it is achieved through a dynamic type system

Polymorphism in practise

- Just overwrite function in the child class

```
function Shape(self, protected)
{
  self.area = function() { throw 'Error'; }
}
function Square(self, protected, width,
height)
{
  define self.area =
      function() { return width * height };
}
```


Class syntax

- A nicer syntax for class definition would be:

```
class Audi inherits Car
{
  Audi(color) : Car(color, 200, 10)
  {
  };
};
```

- Can be achieved with macros (future lecture)

Encapsulation in Python (1/2)

```
class Car:
  def __init__(self, max_speed):
    self.__max_speed = max_speed
  def __update(self):
    ...
c = Car(200)
c._Car__max_speed = 10
c._Car__update()
```

- Protection against 'accident', but still possible to break encapsulation

Encapsulation in Python (2/2)

```
class Car:
  def __init__(self, max_speed):
    _current_speed = 0
    def accelerate():
      nonlocal _current_speed
      _current_speed =
          min(_current_speed + 1,
              max_speed)
    def current_speed():
      return _current_speed
    self.accelerate = accelerate
    self.current_speed = current_speed
```

Closure in C++ pre 11

- If Object-Orientation is defined as +state+closure, does that means that all object-oriented programming language support closure?
- Closure is a reference to a function that has an embedded, persistent, hidden and unseparable context
- The following define a closure (with a state):

```
class sum {
private:
    int m_sum;
public:
    sum() : m_sum(0) {}
    void operator () (int x) { m_sum += x; }
    int get_sum() const { return m_sum; }
}
```


Memory and threading

Memory Management

- Value store is an abstract concept, during execution the value of a binding can be replaced immediately
- With state, comes the need to track when a value is needed or not

Concurrency and state

```
cell x;
thread() {
  x = 1;
}();
thread() {
  x = 3;
}();
```

- No failure, but what is the value of x?
- In some programming languages, can lead to random crashes

Conclusion

- Conscequence of adding state
- Data abstraction
  - Encapsulation
  - Compositionality with polymorphism
  - Instantiation/invocation