Programming (Econometrics)

Lecture 4: Program correctness

Tommi Tervonen

Econometric Institute, Erasmus School of Economics

ERASMUS UNIVERSITEIT ROTTERDAM

Counting sum of the elements of an integer array

```
public static int countSum(int[] array) {
  for (int i=1;i<array.length;i++) {
    array[i] = array[i] + array[i-1];
  }
  return array[array.length-1];
}
```

Counting sum of the elements of an integer array

```
public static int countSum(int[] array) {
  for (int i=1;i<array.length;i++) {
    array[i] = array[i] + array[i-1];
  }
  return array[array.length-1];
}
```

- Returns the correct value, but also modifies the parameter array as a *side effect*.
- What would you expect from:

```
public static int countSum(int[] array)
```

Side effects

- Unexpected side effects make code difficult to understand
- There are also *desired* side effects, e.g. sorting the contents of an array
- In Java we have
  - Accessor methods: returning a value but not modifying contents of the object (`public int getAge()`)
  - Mutator methods: modifying the contents of the object but not returning a value (`public void setAge(int age)`)

Functions and procedures

- In imperative programming we classify methods into
  - **Functions**, that return a value but do not alter the parameters in any way
  - **Procedures**, that alter some of the parameters but do not return a value

```
void setElement(Matrix m, int rInd, int cInd,
  double newElement)
```

```
double getElement(Matrix m, int rInd, int cInd)
```

- Note: if the language does not support exceptions (e.g. C), procedures often do return a value for signifying error conditions

Parameter passing schemes

- For side effects to be possible, parameters have to be passed *by reference*: only a reference (memory address) of the variable is passed to the called method
- Other main technique for parameter passing is to pass *by value*: a local copy of the variable is created within the called method
- Example: pass by reference vs pass by value

Parameter passing in Matlab

- Matlab passes everything by value
- Matrices are passed by reference until they are modified the first time, at which point a local copy is created (!)
- The OO-extension allows to pass references by value by using the handle class

Methods in Matlab

There are no procedures in Matlab: only functions

Methods in Matlab

There are no procedures in Matlab: only functions

- Pros:
  - No undesired side effects
- Cons:
  - No desired side effects
  - Many algorithms can be expressed more clearly with procedures
  - Recursive algorithms become slow without procedures

```
function A = sort(A)
   leftList = A(1:middle);
   rightList = A((middle+1):length(A));
   leftList = sort(leftList);
   rightList = sort(rightList);
  A = merge(leftList, rightList);
end
```

Programming by contract

- Methods define a *contract* between the supplier (you) and the consumer (you or someone else)
- Contract **partially** defined through the signature:

```
function arr = sortArrayFromIndex(array, index)
```

Programming by contract

- Methods define a *contract* between the supplier (you) and the consumer (you or someone else)
- Contract **partially** defined through the signature:

```
function arr = sortArrayFromIndex(array, index)
```

Contract:

1. The `index` has to be in the range `[1, length(array)]` (responsibility of the consumer)
2. If consumer calls the method adhering to (1), then after the method call the following holds: `array[index] < array[index+1] < ... < array[length(array)]` (responsibility of the supplier)

Pre- and post-conditions

```
% Sorts the array in ascending order starting
% from index
%
% PRECOND: 0 < index <= length(array)
% POSTCOND: arr(index) < ...
% ... < arr(length(array))
function arr = sortArrayFromIndex(array, index)
```

- Responsibilities of the consumer are method *pre-conditions* ("Requires")
- Responsibilities of the supplier are method *post-conditions* ("Ensures")
- (PRECOND, METHOD) $\Rightarrow$ POSTCOND

Violating pre-conditions

- As a supplier, if the pre-condition is violated, you are not responsible for what happens
- In practice you should crash the program execution, as the mistake is in the logic

```
function array = sortFromIndex(array, index)
  assert(index > 0 && index <= length(array));
  ... % do the actual sorting
end
```

When to use pre- and post-conditions

- If you cannot handle a possible parameter value, you should declare the accepted range as a pre-conditions
- Post-conditions are often stated in a more informal manner in the method documentations
- Document post-conditions when doing more complex programs, and when you have problems finding bugs

(PRECOND, METHOD) $\Rightarrow$ POSTCOND

How do we know that METHOD ever terminates execution?
How do we know that METHOD does what it's supposed to?

Stop or not?

```
for(i=1:10)
   printf("%d th integer\n", i);
end
```

```
nr = input("How many integers you want?");
for(i=1:nr)
   printf("%d th integer\n", i);
   pause(10*i);
end
```

```
green = true;
while(green)
   green = false;
   pause(10);
   green = true;
end
```

Alan Turing (1912-54)

- Designed the computer that cracked german Enigma in WW2
- Invented LU decomposition
- Invented the Turing test
- Proved the following

The halting problem

Given programs indexed using x, input i, assume that there is a computable function $h(x, i)$:

$$h(x,i) = \begin{cases} 1 & \text{if } x \text{ halts with input } i \\ 0 & \text{otherwise} \end{cases}$$

As we have von Neumann computers (programs $=$ data), we can also call $h(x, x)$

The halting problem

Now, take an arbitrary total computable function f, and construct a partial computable function

$$g(x) = \begin{cases} 0 & \text{if } f(x,x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

g is computable by a program e that loops forever on the undefined case ($\implies$ h is defined on e)

The halting problem

Now, take an arbitrary total computable function f, and construct a partial computable function

$$g(x) = \begin{cases} 0 & \text{if } f(x,x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

g is computable by a program e that loops forever on the undefined case ($\implies$ h is defined on e)

$$\begin{aligned} g(e) = f(e,e) = 0 &\implies h(e,e) = 1 \\ g(e) = \text{ undefined } \implies f(e,e) \neq 0 &\implies h(e,e) = 0 \end{aligned}$$

The halting problem

Now, take an arbitrary total computable function f, and construct a partial computable function

$$g(x) = \begin{cases} 0 & \text{if } f(x,x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

g is computable by a program e that loops forever on the undefined case ($\implies$ h is defined on e)

$$\begin{aligned} g(e) = f(e,e) = 0 &\implies h(e,e) = 1 \\ g(e) = \text{ undefined } \implies f(e,e) \neq 0 &\implies h(e,e) = 0 \end{aligned}$$

$$\implies h \neq f$$

The halting problem

Now, take an arbitrary total computable function f, and construct a partial computable function

$$g(x) = \begin{cases} 0 & \text{if } f(x,x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

g is computable by a program e that loops forever on the undefined case ($\implies$ h is defined on e)

$$\begin{aligned} g(e) = f(e,e) = 0 &\implies h(e,e) = 1 \\ g(e) = \text{ undefined } \implies f(e,e) \neq 0 &\implies h(e,e) = 0 \end{aligned}$$

$$\implies h \neq f$$

$$\implies \text{no such computable function as } h$$

The halting problem

Now, take an arbitrary total computable function f, and construct a partial computable function

$$g(x) = \begin{cases} 0 & \text{if } f(x,x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

g is computable by a program e that loops forever on the undefined case ($\implies$ h is defined on e)

$$\begin{aligned} g(e) = f(e,e) = 0 &\implies h(e,e) = 1 \\ g(e) = \text{ undefined } \implies f(e,e) \neq 0 &\implies h(e,e) = 0 \end{aligned}$$

$$\implies h \neq f$$

$\implies$ no such computable function as h

$\implies$ the halting problem is undecidable

We cannot algorithmically determine whether a program stops execution