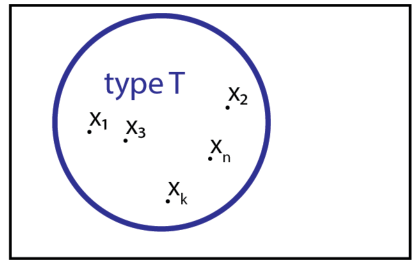
Defensive Programming – Close Examination



1. Given an operation:
   `Operation op1(updates X: T);`

2. Inevitable facts:
   F1. op1 has a contract (i.e., *requires* and *ensures*) whether that contract is explicitly stated or not
   F2. Based on op1's contract, it correctly handles all legal incoming values of parameter x (i.e., all of type T's values), or it cannot

Example

Operation *IncByOne*:

- Handles all values in the range `[min_int..(max_int - 1)]`
- Does not handle the value `max_int`

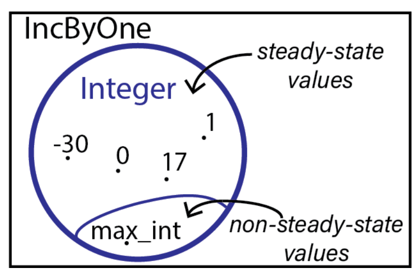
RESOLVE Version

```
Operation IncByOne(updates K: Integer);
requires K < max_int;
ensures K = #K + 1;
Procedure
    K := K + 1;
end IncByOne;
```

C++ Version

```
void incByOne(Integer& k)
//! updates k
//! requires: k < max_int
//! ensures: k = #k + 1
{
   k++;
} // incByOne
```

Terminology



- *steady-state values* – values of type T that can be handled by a called operation
- *non-steady-state values* – values of type T that cannot be handled by a called operation
- *trivial requires clause* – `requires true`
- *non-trivial requires clause* – a *requires* clause that explicitly identifies all steady-state values

3. Consequences of facts that apply to the operation's contract:

C1. If only steady-state values of type T exist, then op1's *requires* clause is trivial

C2. If non-steady-state values of type T exist, then op1 must have a non-trivial *requires* clause

C3. For all steady-state values, the *ensures* clause must state what will be the result of calling op1 on all these values

- Reminder, because of fact F1, these consequences (i.e., C1, C2, and C3) apply whether the contract is explicitly stated or not

4. Design choices for choosing who is responsible for a non-trivial *requires* clause:

O1. The client is responsible for a non-trivial *requires* clause

O2. The called operation (i.e., op1) is responsible for a non-trivial *requires* clause

5. op1's contract in the face of choice O1

- *requires* clause is non-trivial
- *ensures* clause states one thing:
  1) how steady-state incoming values will be processed into outgoing values

6. op1's contract in the face of choice O2

- *requires* clause is non-trivial (keep the *requires* clause non-trivial, see Note below)
- *ensures* clause must state two things:
  1) how steady-state incoming values will be processed into outgoing values
  2) what will happen when the incoming are non-steady-state values
- Note:
  - This choice is often referred to as "defensive programming"
  - Is recommended that you keep the non-trivial version of the *requires* clause as an advertisement to the calling operation what are the steady-state values

7. Defensive programming design choices (choice O2 above)

7.1 Do nothing when incoming is non-steady-state data

- *ensures* clause must state two things:
  1) steady-state incoming values are processed normally
  2) non-steady-state incoming results in: x = #x
- Problem: Often there is no way to implement the calling operation so that it can detect when a "do nothing" was executed by Op1. This is because the results of some of the steady-state data will look exactly the same as a "do nothing".

```
void incByOne(Integer& k)
//! updates k
//! requires:   k < max_int
//! ensures:    ((k < max_int) -> (k = #k + 1))
//!             and (~(k < max_int) -> (k = #k))
{
   if (k < INT_MAX) {
      k++;
   } // end if
} // incByOne
```

7.2 Produce true/false when incoming is non-steady-state data

- *ensures* clause must state two things:
  1) steady-state incoming processed are normally and *true* is produce back to the caller
  2) non-steady-state incoming results in: x = #x and *false* is produced back to the caller
- Note:
  - An additional Boolean parameter will need to be added to op1's parameter list
  - If you are going to write a defensive operation, i.e., one that checks its preconditions, then it is recommended that you have the operation return a Boolean *successful* for signaling which was handled by the operation, steady-state data or non-steady-state data

```
void incByOne(Integer& k, Boolean& successful)
//! updates k, successful
//! requires:   k < max_int
//! ensures:    ((k < max_int) -> (k = #k + 1 and successful))
//!             and (~(k < max_int) -> (k = #k and ~successful))
{
successful = (k < INT_MAX)
if (successful) {
    k++;
} // end if
} // incByOne
```

7.3 Throw an exception when incoming is non-steady-state data

- *ensures* clause must state two things:
  1) steady-state incoming processed normally and *true* is produce back to the caller
  2) non-steady-state incoming results in: x = #x and an exception being thrown
- Note:
  - Many design guidelines recommend that the exception throwing approach only be used for situations that are outside the software system's sphere of control, e.g., a failure when trying to open or access an internet connection, or a file
  - There is currently no good way to mathematically capture all the aspects of exception throwing and handling, so the *ensures* clause (below) is not pure mathematics, it is a mixture of math and programming concepts

```
void incByOne(Integer& k)
//! updates k
//! requires:   k < max_int
//! ensures:    ((k < max_int) -> (k = #k + 1))
//!             and (~(k < max_int) -> throws exception)
{
if (k < INT_MAX) {
    k++;
} else {
    throw exception();
} // end if
} // incByOne
```