

# SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2017

## 11 Generic MIS (Ghezzi Ch. 4)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

January 31, 2017



# Generic MIS

- Administrative details
- Two useful LaTeX packages
- `ssh -X`
- Uses for abstract objects
- Generic modules
- Generic stack abstract object
- Properties exhibited by a stack module
- Access routine idioms
  - ▶ Set idioms
  - ▶ Sequence idioms
  - ▶ Tuple idioms
- Exceptions
- Quality Criteria
- Generic queue ADT

# Administrative Details

- Assignment 1

- ▶ Files due by 11:59 pm January 28
- ▶ E-mail partner files by 11:59 pm January 29
- ▶ Lab report due by 11:59 pm February 2
- ▶ Using Python 2.7, doxygen, make, LaTeX, git
- ▶ Make sure everything runs on mills
- ▶ Use the folder structure and filenames given!
- ▶ Commit your doxygen configuration file
- ▶ Do NOT commit generated html and latex folders
- ▶ Generate the documentation in the A1 folder
- ▶ Commit the report.pdf file
- ▶ Can use “private” methods (`__methodName__`)
- ▶ You might find the listings and hyperref packages useful
- ▶ If you know how, tag your Jan 28 submission A1Sub
- ▶ Marking key

# Administrative Details

- Remember that Avenue has a search feature for the Discussion board
- For Assignment 2, try to use git for version control, as opposed to thinking of it as a submission system
- Assignment 2
  - ▶ Files due by 11:59 pm Feb 15
  - ▶ E-mail partner files by 11:59 pm Feb 16
  - ▶ Lab report due by 11:59 pm Feb 27
- Benches versus work surface?

# Listings and Hyperref Packages

- [https://en.wikibooks.org/wiki/LaTeX/Source\\_Code\\_Listings](https://en.wikibooks.org/wiki/LaTeX/Source_Code_Listings)
- <https://www.sharelatex.com/learn/Hyperlinks>
- See an example [here](#)

# X Windows

- If you want to use a gui on mills, you can use `ssh -X`
- `ssh -X mills.cas.mcmaster.ca`
- On a Mac you need XQuartz installed
- If you know the details for Windows or Linux, please post to Avenue

# Uses for Abstract Objects

- Creating a single abstract object corresponds to the singleton design pattern
- Provides “global” variables
- Uses
  - ▶ Shared resource
  - ▶ Hoffman and Strooper example (assembler)
  - ▶ Look up table for global state (read only)
  - ▶ Logger (write only)
- Problematic for testing if high coupling and frequent state changes, since many test cases will depend on the state of the abstract object

# Generic Modules

- What if we have a sequence of integers, instead of a sequence of point masses?
- What if we want a stack of integers, or characters, or pointT, or pointMassT?
- Do we need a new specification for each new abstract object?
- No, we can have a single abstract specification implementing a family of abstract objects that are distinguished only by a few variabilities
- Rather than duplicate nearly identical modules, we parameterize one **generic module** with respect to type(s)
- Advantages
  - ▶ Eliminate chance of inconsistencies between modules
  - ▶ Localize effects of possible modifications
  - ▶ Reuse



# Generic Stack Module Syntax

## Generic Module

Stack(T)

## Exported Constants

MAX\_SIZE = 100

## Exported Access Programs

Routine name	In	Out	Exceptions
...	...	...	...

# Stack Module Syntax

## Exported Access Programs

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
s_init			
s_push	T		FULL
s_pop			EMPTY
s_top		T	EMPTY
s_depth		integer	

# Semantics

**State Variables**

**State Invariant**

**Assumptions**

# Semantics

## State Variables

$s$ : sequence of  $T$

## State Invariant

## Assumptions

# Semantics

## State Variables

$s$ : sequence of  $T$

## State Invariant

$$|s| \leq \text{MAX\_SIZE}$$

## Assumptions

# Semantics

## State Variables

$s$ : sequence of  $T$

## State Invariant

$$|s| \leq \text{MAX\_SIZE}$$

## Assumptions

$s\_init()$  is called before any other access routine

# Access Routine Semantics

s\_init():

- transition:
- exception:

s\_push(x):

- transition:
- exception:

s\_pop():

- transition:
- exception:

# Access Routine Semantics

s\_init():

- transition:  $s := \langle \rangle$
- exception:

s\_push(x):

- transition:
- exception:

s\_pop():

- transition:
- exception:



# Access Routine Semantics

s\_init():

- transition:  $s := \langle \rangle$
- exception: none

s\_push(x):

- transition:
- exception:

s\_pop():

- transition:
- exception:

# Access Routine Semantics

s\_init():

- transition:  $s := \langle \rangle$
- exception: none

s\_push(x):

- transition:  $s := s || \langle x \rangle$
- exception:

s\_pop():

- transition:
- exception:

# Access Routine Semantics

`s_init()`:

- transition:  $s := \langle \rangle$
- exception: none

`s_push(x)`:

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (|s| = MAX\_SIZE \Rightarrow FULL)$

`s_pop()`:

- transition:
- exception:

# Access Routine Semantics

`s_init()`:

- transition:  $s := \langle \rangle$
- exception: none

`s_push(x)`:

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (|s| = MAX\_SIZE \Rightarrow FULL)$

`s_pop()`:

- transition:  $s := s[0..|s| - 2]$
- exception:

# Access Routine Semantics

`s_init()`:

- transition:  $s := \langle \rangle$
- exception: none

`s_push(x)`:

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

`s_pop()`:

- transition:  $s := s[0..|s| - 2]$
- exception:  $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

# Access Routine Semantics Continued

s\_top():

- output:
- exception:

s\_depth():

- output:
- exception:

# Access Routine Semantics Continued

`s_top()`:

- output: *out* :=  $s[|s| - 1]$
- exception:

`s_depth()`:

- output:
- exception:

# Access Routine Semantics Continued

`s_top()`:

- output:  $out := s[|s| - 1]$
- exception:  $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

`s_depth()`:

- output:
- exception:



# Access Routine Semantics Continued

`s_top()`:

- output:  $out := s[|s| - 1]$
- exception:  $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

`s_depth()`:

- output:  $out := |s|$
- exception:

# Access Routine Semantics Continued

`s_top()`:

- output:  $out := s[|s| - 1]$
- exception:  $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

`s_depth()`:

- output:  $out := |s|$
- exception: `none`

# Stack Module Properties

$\{true\}$   
     $s\_init()$   
 $\{|s'| = 0\}$

$\{|s| < MAX\_SIZE\}$   
     $s\_push(x)$   
 $\{|s'| = |s| + 1 \wedge s'[|s'| - 1] = x \wedge s'[0..|s| - 1] = s[0..|s| - 1]\}$

$\{|s| < MAX\_SIZE\}$   
     $s\_push(x)$   
     $s\_pop()$   
 $s' = s$

# Set Idiom

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
set_add	T		Member, Full
set_del	T		NotMember
set_member	T	boolean	
set_size		integer	

# Sequence Idiom

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
seq_init			
seq_add	integer, T		PosOutOfRange, Full
seq_del	integer		PosOutOfRange
seq_setval	integer, T		PosOutOfRange
seq_getval	integer	T	PosOutOfRange
seq_size		integer	
seq_start			
seq_next		T	AtEnd
seq_end		boolean	
seq_append	T		Full

# Tuple Idiom Version 1

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
tp_init			
tp_set_f <sub>1</sub>	T <sub>1</sub>		
tp_get_f <sub>1</sub>		T <sub>1</sub>	
...	...	...	...
tp_set_f <sub>N</sub>	T <sub>N</sub>		
tp_get_f <sub>N</sub>		T <sub>N</sub>	

## Tuple Idiom Version 2

<b>Routine name</b>	<b>In</b>	<b>Out</b>	<b>Exceptions</b>
tp_init			
tp_set	$T_1, T_2, \dots, T_N$		
tp_get		$T$	

# Exception Signaling

- Useful to think about exceptions in the design process
- Will need to decide how exception signalling will be done
  - ▶ A special return value, a special status parameter, a global variable
  - ▶ Invoking an exception procedure
  - ▶ Using built-in language constructs
- Caused by errors made by programmers, not by users
- Write code so that it avoid exceptions
- Exceptions will be particularly useful during testing



# Assumptions versus Exceptions

- The assumptions section lists assumptions the module developer is permitted to make about the programmer's behaviour
- Assumptions are expressed in prose
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation
- Interface design should provide the programmer with a means to check so that they can avoid exceptions
- When an exceptions occurs no state transitions should take place, any output is *don't care*

# Quality Criteria

- Consistent
  - ▶ Name conventions
  - ▶ Ordering of parameters in argument lists
  - ▶ Exception handling, etc.
- Essential - omit unnecessary features
- General - cannot always predict how the module will be used
- As implementation independent as possible
- Minimal - avoid access routines with two potentially independent services
- High cohesion - components are closely related
- Low coupling - not strongly dependent on other modules
- Opaque - information hiding

# Generic Queue Module ADT Syntax

## Generic Template Module

QueueADT(T)

## Exported Types

queueT = ?

## Exported Constants

MAX\_SIZE = 100

## Exported Access Programs

Routine name	In	Out	Exceptions
...	...	...	...

# QueueADT Module Syntax

## Exported Access Programs

Routine name	In	Out	Exceptions
q_init		queueT	
add	T		queue_not_init, queue_full
pop			queue_not_init, queue_empty
front		T	queue_not_init, queue_empty
isempty		boolean	queue_not_init

# Semantics

**State Variables**

**State Invariant**

**Assumptions**

# Semantics

## State Variables

$s$ : sequence of  $T$

$is\_init$  : boolean := *false*

## State Invariant

## Assumptions

# Semantics

## State Variables

$s$ : sequence of  $T$

$is\_init$  : boolean  $:= false$

## State Invariant

$|s| \leq MAX\_SIZE$

## Assumptions

# Access Routine Semantics

q\_init():

- transition:
- output:
- exception:

add(x):

- transition:
- exception:

pop():

- transition:
- exception:



# Access Routine Semantics

q\_init():

- transition:  $s, is\_init := \langle \rangle, true$
- output:  $out := self$
- exception: `none`

add(x):

- transition:
- exception:

pop():

- transition:
- exception:

# Access Routine Semantics

q\_init():

- transition:  $s, is\_init := \langle \rangle, true$
- output:  $out := self$
- exception: none

add(x):

- transition:  $s := s || \langle x \rangle$
- exception:

pop():

- transition:
- exception:

# Access Routine Semantics

q\_init():

- transition:  $s, is\_init := \langle \rangle, true$
- output:  $out := self$
- exception: none

add(x):

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = MAX\_SIZE \Rightarrow queue\_full)$

pop():

- transition:
- exception:

# Access Routine Semantics

q\_init():

- transition:  $s, is\_init := \langle \rangle, true$
- output:  $out := self$
- exception: none

add(x):

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = MAX\_SIZE \Rightarrow queue\_full)$

pop():

- transition:  $s := s[1..|s| - 1]$
- exception:

# Access Routine Semantics

q\_init():

- transition:  $s, is\_init := \langle \rangle, true$
- output:  $out := self$
- exception: none

add(x):

- transition:  $s := s || \langle x \rangle$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = MAX\_SIZE \Rightarrow queue\_full)$

pop():

- transition:  $s := s[1..|s| - 1]$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = 0 \Rightarrow queue\_empty)$

# Access Routine Semantics Continued

front():

- output:
- exception:

isempty():

- output:
- exception:

# Access Routine Semantics Continued

front():

- output: *out* := *s*[0]
- exception:

isempty():

- output:
- exception:

# Access Routine Semantics Continued

front():

- output:  $out := s[0]$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = 0 \Rightarrow queue\_empty)$

isempty():

- output:
- exception:



# Access Routine Semantics Continued

front():

- output:  $out := s[0]$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = 0 \Rightarrow queue\_empty)$

isempty():

- output:  $out := |s| = 0$
- exception:

# Access Routine Semantics Continued

front():

- output:  $out := s[0]$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init \mid |s| = 0 \Rightarrow queue\_empty)$

isempty():

- output:  $out := |s| = 0$
- exception:  $exc := (\neg is\_init \Rightarrow queue\_not\_init)$

# Queue Module Properties

$\{true\}$

$q\_init()$

$\{|s'| = 0 \wedge is\_init\}$

$\{|s| < MAX\_SIZE\}$

$add(x)$

$\{|s'| = |s| + 1 \wedge s'[0] = x \wedge s'[1..|s'| - 1] = s[0..|s| - 1]\}$