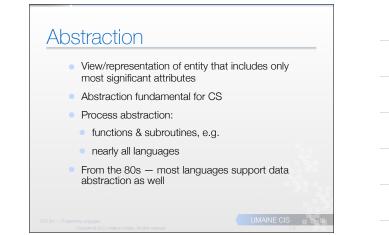


- Chapters 11 & 12 in the book
- Slides are heavily based on Sebesta's slides for the chapters, with much left out!

## Abstract data types &

- The Concept of Abstraction
- Introduction to Data Abstraction
- Design Issues for Abstract Data Types
- Language Examples
- Parameterized Abstract Data Types
- Encapsulation Constructs
- Naming Encapsulations





# Abstract data type (ADT)

- Abstract data type: class of data types defined by a set of values and behavior/operations
- E.g., lists, queues, stacks...
- Sometimes: includes time complexity in definition
- With respect to a programming language: userdefined data type that:
  - hides the representation of "objects" only operations possible are provided by the type
  - single syntactic unit contains the declarations of the type and of any operations on it

### Advantages

- Advantages of hiding data:
  - reliability: user code can't access internals, thus compromising other users' use of object
  - flexibility: since user code can't access internals, internals can be changed to improve performance w/o affecting users
  - reduced name conflicts
- Advantages having single syntactic unit for type:
- Provides way to organize program
- Enhances modifiability: everything needed for data structure is together in one place
- Separate compilation, debugging

COS 301 — Programming Langu

VIAINE CIS

## ADT language requirements

- Syntactic unit for encapsulating definition
- Way to make type names, method/subprogram headers available while hiding definitions
- Primitive operations on types must be part of the compiler/interpreter

### Design issues

- What does the container for the interface to the type look like?
- Can abstract types be parameterized?
- What access controls are provided?
- Is specification of the type separate from its implementation?

COS 301 — Programming Lang

UMAINE CIS

### Language example: Ada

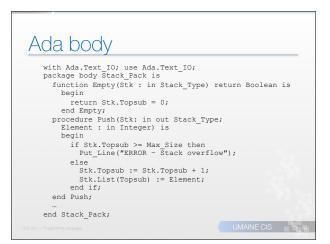
- Encapsulation construct: package
  - Interface: specification package
  - Implementation: body package
- Information hiding <u>public</u> and <u>private</u> parts of specification package
  - Public part: name, maybe representation of any unhidden types
  - Private part:
    - representation of the abstract type
    - private types have built-in operations for assignment, comparison
    - limited private types have no built-in operations

05 301 — Programming Language

# Ada specification

package Stack\_Pack is type stack\_type is limited private; max\_size: constant := 100; function empty(stk: in stack\_type) return Boolean; procedure push(stk: in out stack\_type; elem: in Integer); procedure pop(stk: in out stack\_type); function top(stk: in stack\_type) return Integer; // private -- hidden from clients type list\_type is array (1..max\_size) of Integer; type stack\_type is record

list: list\_type; topsub: Integer range 0..max\_size) := 0; end record; end Stack\_Pack



## C++ example

- Encapsulation is via classes
- ADT based on C struct, Simula 67 class
- Classes are types
- All instances of a class share copy of member • functions (methods)
- Each instance has its own copy of class data members (instance variables)
- Instances can be static, stack dynamic, or heap dynamic

### C++ example

- Information hiding:
  - Private clause for hidden entities
  - Public clause for interface entities
  - Protected clause for inheritance (later)

#### Constructors:

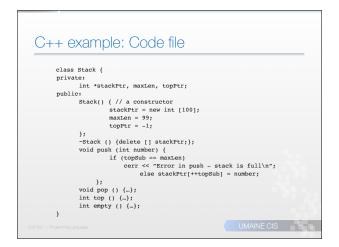
- Functions to initialize the data members they don't create objects
- May also allocate storage if part of the object is heap-dynamic
- Can include parameters to provide parameterization of the objects
- Implicitly called when an instance is created but can be called explicitly, too Name is the same as the class name

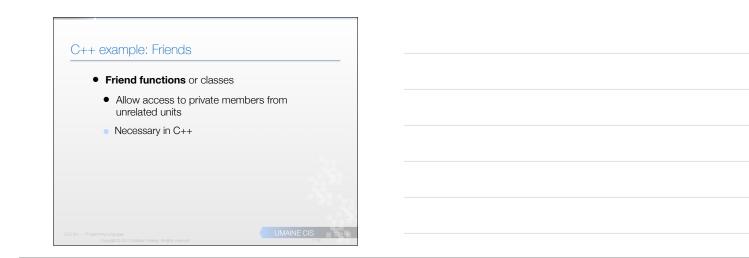
#### Destructors:

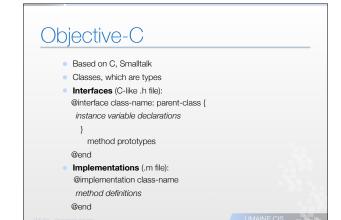
- Clean up after an instance is destroyed usually just to reclaim heap storage Implicitly called when the object's lifetime ends, or explicitly called Name is the class name, preceded by a tilde (~)

#### C++ example: Header file

// Stack.h - the header file for the Stack class #include <iostream.h> class Stack {
private: //\*\* These members are visible only to other //\*\* members and "friends" (see textbook) int \*stackPtr; int maxLen; vublic: //\*\* These members are visible to clients Stack(); //\*\* A constructor -Stack(); //\*\* A destructor int topPtr; public: void push(int); void pop(); int top(); int empty(); } UMAINE CIS







# **Objective-C example**

#### Method prototypes

- (+ I -) (return-type) method-name [: (formal-parameters)];
- +/- for class/instance methods (resp.)
- Colon, parentheses not included when no parameters
- Odd nomenclature:
  - One parameter:
  - Ex: (int) foo: (int) x;
  - Name of method is foo:
  - **Message**: (call): [objectName foo: 3]  $\rightarrow x = 3$
  - Two parameters:
  - Ex: (int) foo: (int) x bar: (float) y;
  - Name of method is foo:bar:
  - Message: [objectName foo: 3 bar: 4.5]  $\rightarrow$  x = 3, y = 4.5
- **Objective-C example** • Initializers: constructors Only initialize variables Can have any name, and are only explicitly called Initializers return the instance itself • Create object → call alloc + initializer Adder \*myAdder = [[Adder alloc] init];
  - All class instances are heap dynamic

# **Objective-C example**

- Standard prototypes (e.g., for I/O):
  - #import <Foundation/Foundation.h>
- Program must initialize a **pool** for its storage:
- NSAutoreleasePool \*pool = [[NSAutoreleasePool alloc] init];
- NSxxx from NextStep
- At program end, release storage: [pool drain];

UMAINE CIS

# Objective-C — information

- @public, @private, @protected specify instance variable access
  - @public: accessible anywhere
  - @private: accessible only in class where defined
  - @protected: accessible in that class and any subclasses
  - Default access is @protected
- However: no really good way to restrict access to methods
- Getter and setter methods for instance variables
  - Name of getter is always name of instance variable
  - Name of setter is always the word set with the capitalized variable name attached (e.g., setFoo)
  - Can be implicitly generated if no additional constraints to be defined — called "properties" in this case

01 — Programming Languages

# Objective-C – another

// stack.m – interface and implementation for simple stack	eimplementation Stack
<pre>#import <foundation foundation.h=""> @interface Stack: NSObject {     int stackArray[100], stackPtr,maxLen, topSic     .</foundation></pre>	<pre>-(Stack *) initWith {     maxLen = 100;     topSub = -1;     stackPtr = stackArray;     return self;</pre>
} -(void) push: (int) number; -(void) pop; -(int) top; -(int) empty; @end	<pre>} } -(void) push: (int) number {     if (topSub == maxLen)         NSLog(@"Stack is full");     else         stackPtr[++topSub] = number; } @end</pre>
	UMAINE CIS

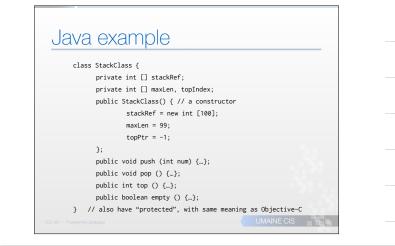
Using the stack ADT		
int main (int argc, char *argv[]) {		
int temp;		
NSAutoreleasePool *pool = [[NSAutoreleasePool al	loc] init];	
<pre>Stack *myStack = [[Stack alloc] initWith];</pre>		
[myStack push: 5];		
[myStack push: 3];		
<pre>temp = [myStack top];</pre>		
<pre>NSLog(@"Top element is: %i", temp);</pre>		
[myStack pop];		
<pre>temp = [myStack top];</pre>		
<pre>NSLog(@"Top element is: %i", temp);</pre>		
<pre>temp = [myStack top];</pre>		
myStack pop];		
[myStack release];		
[pool drain];		
return 0;		
3	UMAINE CIS	and the second second

### Java

- Similar to C++, except:
  - All user-defined types are classes
  - All objects are heap-dynamic
  - All objects accessed via reference variables
  - Access control modifiers for class entities
  - Package scope:
    - All entities in all classes in package that are not restricted by access control modifiers → visible throughout package
  - Eliminates need for C++'s friend functions & classes

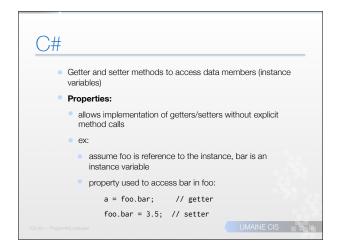
COS 301 — Programming Lang

UMAINE CIS



### C#

- Based on C++, Java
- Adds two access modifiers, internal (within project) and protected internal (= protected or internal)
- All class instances: heap dynamic
- Default constructors available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- structs are lightweight classes that do not support inheritance



### Ruby

#### Encapsulation construct: class

- Variable names:
- Local: regular identifiers
- Instance variables: begin with @
- Class variables: begin with @@
- Methods: defined with function definition syntax (def...end)
- Constructors:
  - Named initialize
  - Only one per class
  - Implicitly called when new is called
  - If additional constructors needed: different names, and they must call new
- Class members can be marked private or public (default)

Classes are heap dynamic

COS 301 — Programming Languag

UMAINE CIS



# Parameterized ADTs

#### Parameterized ADTs

- can design an ADT to store any element type (e.g.)
- only issue for statically-typed languages
- Also known as generic classes
- Supported in C++, Ada, Java (5.0), C# (2005)

COS 301 — Programming Language

<pre>generic Max_Size: Positive; type Elem_Type is private; package Generic_Stack is type Stack_Type is limited private; function Empty(Stk : in Stack_Type) return Boolean; function Top(Stk: in out StackType) return Elem_type;  private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record; end Generic_Stack; package Integer_Stack is new Generic_Stack(100,Integer)</pre>		
<pre>type Elem_Type is private; package Generic_Stack is type Stack_Type is limited private; function Empty(Stk: in Stack_Type) return Boolean; function Top(Stk: in out StackType) return Elem_type;  private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	generic	
<pre>package Generic_Stack is type Stack_Type is limited private; function Empty(Stk : in Stack_Type) return Boolean; function Top(Stk: in out StackType) return Elem_type; private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record; </pre>	Max_Size: Positive;	
<pre>type Stack_Type is limited private; function Empty(Stk : in Stack_Type) return Boolean; function Top(Stk: in out StackType) return Elem_type;  private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	type Elem_Type is priv	/ate;
<pre>function Empty(Stk : in Stack_Type) return Boolean; function Top(Stk: in out StackType) return Elem_type;  private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	package Generic_Stack is	3
<pre>function Top(Stk: in out StackType) return Elem_type;  private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	type Stack_Type is lim	nited private;
<pre>private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	function Empty(Stk : i	in Stack_Type) return Boolean;
<pre>private type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>	function Top(Stk: in c	out StackType) return Elem_type;
<pre>type List_Type is array (1Max_Size) of Element_Type; type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;</pre>		
type Stack_Type is record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;	private	
record List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;	type List_Type is arra	ay (1Max_Size) of Element_Type;
List : List_Type; Topsub : Integer range 0 Max_Size := 0; end record;	type Stack_Type is	
Topsub : Integer range 0 Max_Size := 0; end record;	record	
end record;	List : List_Type;	
	Topsub : Integer rar	nge 0 Max_Size := 0;
end Generic_Stack; package Integer_Stack is new Generic_Stack(100,Integer	end record;	
	end Generic_Stack;	package Integer_Stack is new Generic_Stack(100,Intege

# Parameterized ADTs in C++

<ul> <li>Can make classes somewhat generic with parameterized constructors:</li> </ul>	
<pre>Stack (int size) {     stk_ptr = new int [size];     max_len = size - 1;     top = -1;</pre>	
};	
<pre>Stack stk(150);</pre>	

UMAINE CIS

<pre>template <class type=""> class Stack {     private:     Type *stackPtr;     const int maxLen;     int topPtr;     public:     Stack() f // Constructor for 100 elements     stackOPtr = new Type[100];     maxLen = 99;     topPtr = -1;     Stack(int size) { // Constructor for a given number     stackPtr = new Type[size];     maxLen = size - 1;     topSub = -1;     }     _     _ } </class></pre>		s in C++ — templates
<pre>Type *stackPtr; const int maxLen; int topPtr; yublic: Stack() { // Constructor for 100 elements stackPtr = new Type[100]; maxLen = 99; topPtr = -1; } Stack(int size) { // Constructor for a given number stackPtr = new Type[size]; maxLen = size - 1; topSub = -1; }</pre>		
<pre>int topPtr; public: Stack() { // Constructor for 100 elements stackPtr = new Type[100]; maxLen = 99; topPtr = -1; } Stack(int size) { // Constructor for a given number stackPtr = new Type[size]; maxLen = size - 1; topSub = -1; } </pre>	Type *stackPtr;	
<pre>Stack() { // Constructor for 100 elements stackPrr = new Type[100]; maxLen = 99; topPtr = -1; } Stack(int size) { // Constructor for a given number stackPtr = new Type[size]; maxLen = size - 1; topSub = -1; }</pre>	int topPtr;	
<pre>stackPtr = new Type[size]; maxLen = size - 1; topSub = -1; }</pre>	<pre>Stack() { // Constructo stackPtr = new Type[10 maxLen = 99; topPtr = -1:</pre>	yr for 100 elements M0];
<pre>maxLen = size - 1; topSub = -1; </pre>	Stack(int size) { // Con	structor for a given number
topSub = -1; }	stackPtr = new Type[si	ze];
)	maxLen = size - 1;	
-	topSub = -1;	
	}	
	}	

### Encapsulation constructs

- Large programs two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of partial compilation i.e., compilation units smaller than whole program
- $\Rightarrow$  Group logically-related subprograms into units
- Allow units to be separately compiled (i.e., compilation units)
- Such units are encapsulation constructs

UMAINE CIS

### Nested subprograms as encapsulation

- One way to organize subprograms: nest them
- Inner subprograms are encapsulated within outer, but can share variables
- Supported in Ada, Fortran 95+, Python, JavaScript, Ruby, Lisp, ...

# Encapsulation in C

- Encapsulation in C basically a compilation unit
- Interface is placed (by convention) in header (.h) file
- Implementation in .c file
- #include used to include header files
- Problem: linker doesn't check types between header and implementation

COS 301 - Programming Lang

UMAINE CIS

# Encapsulation in C++

- Header & code files, like C
- Also has classes
  - Class definition used as the interface
  - Member (instance variables, methods) defined in separate file
- Friend functions/classes provide a way to grant access to private members of a class

#### COS 301 — Programming Languages

## Encapsulation in Ada

- Packages encapsulation unit in Ada
- Specification packages any number of data, subprogram definitions
- Specification, body parts of package can be compiled separately

#### Encapsulation in C#

- Assembly: collection of files that appears as a single
- executable or...
- ...dynamic link library (DLL)
  - Microsoft's version of shared libraries
  - collection of classes, methods (in C#) that are individually linked to an executing program
- Each file contains module that can be separately compiled
- **Internal** access modifier: member is visible to all classes in the assembly

## Naming encapsulations

- Large programs:
  - define many global names
  - need way to divide into logical groups
- Naming encapsulation: used to create a new scope for names
- C++ namespaces
- Can place each library in its own namespace
- Qualify names used outside with their namespace, e.g., foo::bar, foo::baz
- C# also includes namespaces

UMAINE CIS

# Naming encapsulations

- Java packages
  - Package contains one or more class definitions
  - Classes within package are partial friends
  - Clients of a package use fully qualified name or use the **import** declaration
- Ada packages
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the with clause

S 301 — Programming Languages

## Naming encapsulations

Ruby:

- Classes, but also modules
- Typically encapsulate collections of constants and methods
- Modules cannot be instantiated or subclassed, and they cannot define variables
- Methods defined in a module must include the module's name
- Access to the contents of a module is requested with the **require** method

COS 301 — Programming Langue