Principles of Programming Languages
Lecture 17

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

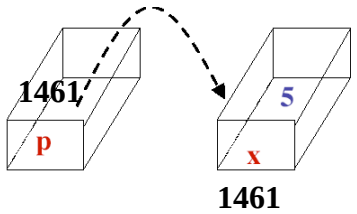
Pointers

- **Variables referencing memory address or NIL/NULL**
  - PL/I is the first high-level language to have pointer variables

- **Operations:**
  - Assignment to memory address (allocation)
    - Note that this could be done with/without allocation
    - E.g. // C lang

```
int b;
int *a = &b;
int *a = (int *) malloc(sizeof(int));
```

```
int *p;
p = &x;
*p = 5;
```



  - Reference to value stored in memory cell
    - E.g.

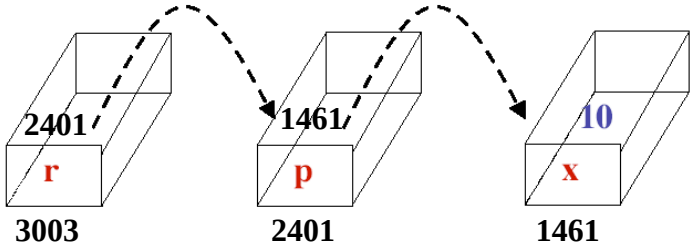
```
int b;
b = *a;
```

  - Release of memory address (de-allocation)
    - Ada, ALGOL 68: no explicit de-allocation

Pointers cont'd

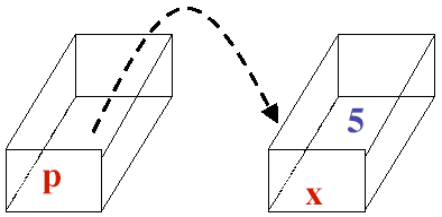
- **Implementation:**
  - Usually 2 or 4 bytes
  - Hardware restrictions (e.g. Intel architecture)
  - Note that you can have pointer to pointer… to value

```
int *p;
int ** r;
p = &x;
*p = 5;
r = &p;
**r = 10;
```



Pointers cont'd

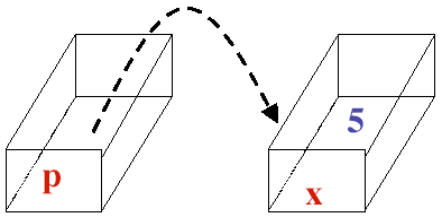
- **Why are pointers problematic?**
  - Type checking:
    - PL/I allowed pointers to point to any type of object!



What can we do about pointer problems?

Pointers cont'd

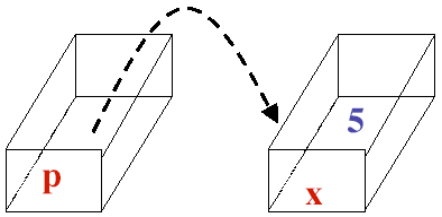
- **Why are pointers problematic?**
  - Type checking:
    - PL/I allowed pointers to point to any type of object!
  - Dangling Reference:
    - Storage pointed to is freed, but pointer is not set to null.
    - Then, you are able to access storage whose value are not meaningful.



What can we do about pointer problems?

Pointers cont'd

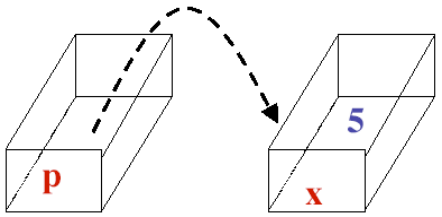
- **Why are pointers problematic?**
  - Type checking:
    - PL/I allowed pointers to point to any type of object!
  - Dangling Reference:
    - Storage pointed to is freed, but pointer is not set to null.
    - Then, you are able to access storage whose value are not meaningful.
  - Garbage:
    - Pointer itself is freed (perhaps by execution going out of scope) but heap locations pointed to are not freed
    - Then, there is no way to access this heap storage



What can we do about pointer problems?

Pointers cont'd

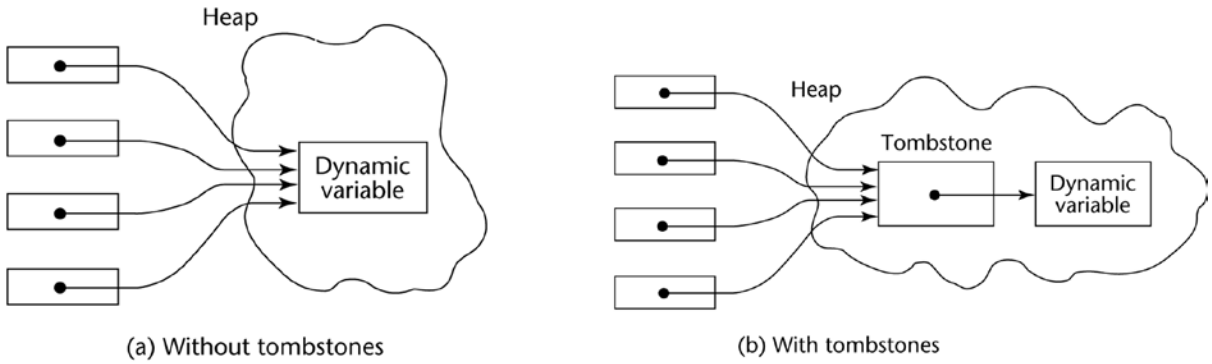
- **Why are pointers problematic?**
  - Type checking:
    - PL/I allowed pointers to point to any type of object!
  - Dangling Reference:
    - Storage pointed to is freed, but pointer is not set to null.
    - Then, you are able to access storage whose value are not meaningful.
  - Garbage:
    - Pointer itself is freed (perhaps by execution going out of scope) but heap locations pointed to are not freed
    - Then, there is no way to access this heap storage
  - Memory leaks:
    - Gradual loss of available computer memory when a program repeatedly fails to return memory that it has obtained for temporary use.
    - Then, the available memory for that application becomes exhausted and the program can no longer function.



What can we do about pointer problems?

Pointers cont'd

- **Tombstones:**
  - Add an extra memory location that points to the value
  - Pointers only point to tombstones, never to value
  - When pointer is de-allocated, do not delete tombstone
  - Problems:
    - Expensive in time because of extra indirection
    - Expensive in space because they are never deleted **until** program exists!

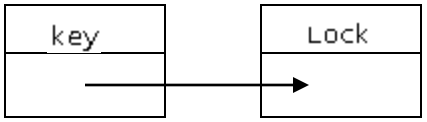


(a) Without tombstones

(b) With tombstones

Pointers cont'd

- **Locks-and-keys:**
  - Associate a key with the pointer and a lock with corresponding value
  - Access is granted if key match lock



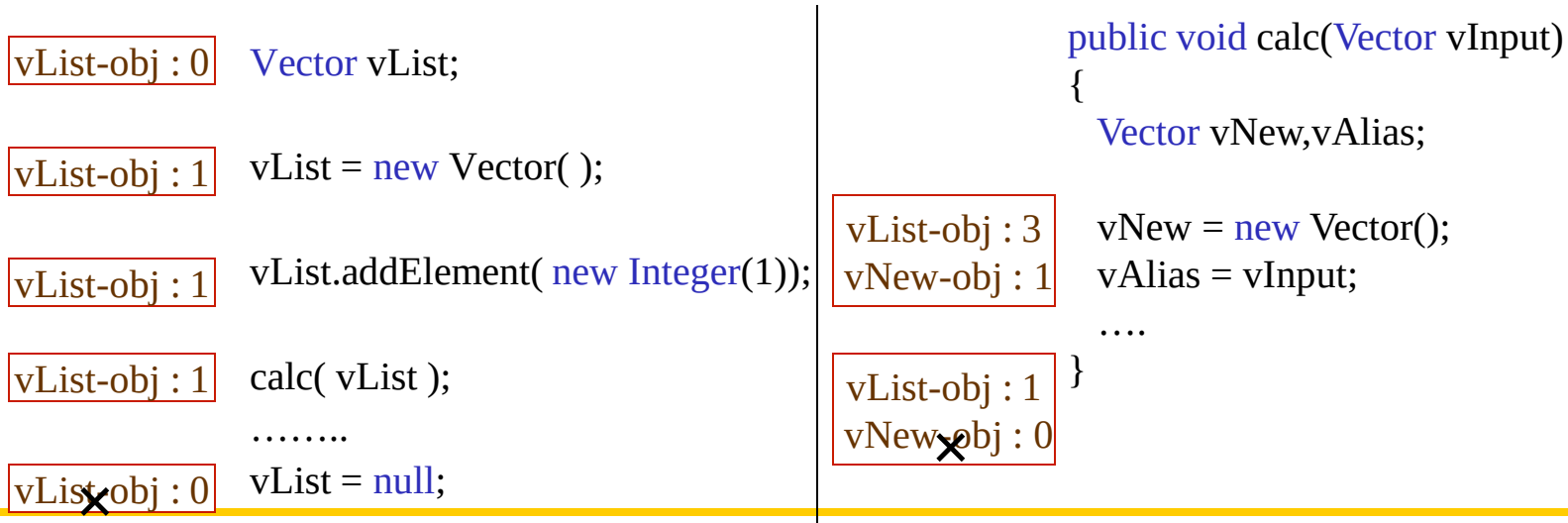

```
int * pInt;
int X  = 10;
pInt = &X;          // this will succeed if lock of X and that of pInt match
```

Pointers cont'd

- **Garbage Collection(GC):**
  - An automatic memory management scheme implemented by the runtime environment
  - Analyzes usage of memory and recover pieces of storage no longer reachable from user pointers and references
  - Pros:
    - Simplify programming,
    - Shorten development lifecycle (less memory problems…)
  - Cons:
    - Execution time cost traded for easier job for user
    - Unsuitable for real time systems
  - E.g.:
    - Java, ML, Ada, Modula, Python
  - Many gc algorithms (*active research area*)

Pointers cont'd

- **GC Algorithms:**
  - Reference Counting (Smart Pointers):
    - Maintain total number of pointers to a storage block
      - Each variable has an additional attribute (a counter) telling how many pointers are pointing to that variable.
      - Every time a pointer is disconnected, decrement counter by 1 and check for 0
      - Every time a pointer is connected, increment the counter by 1
      - If counter is 0, delete the variable.
    - Problems:
      - Costs extra memory and execution time for updates. Circular references
    - Example:

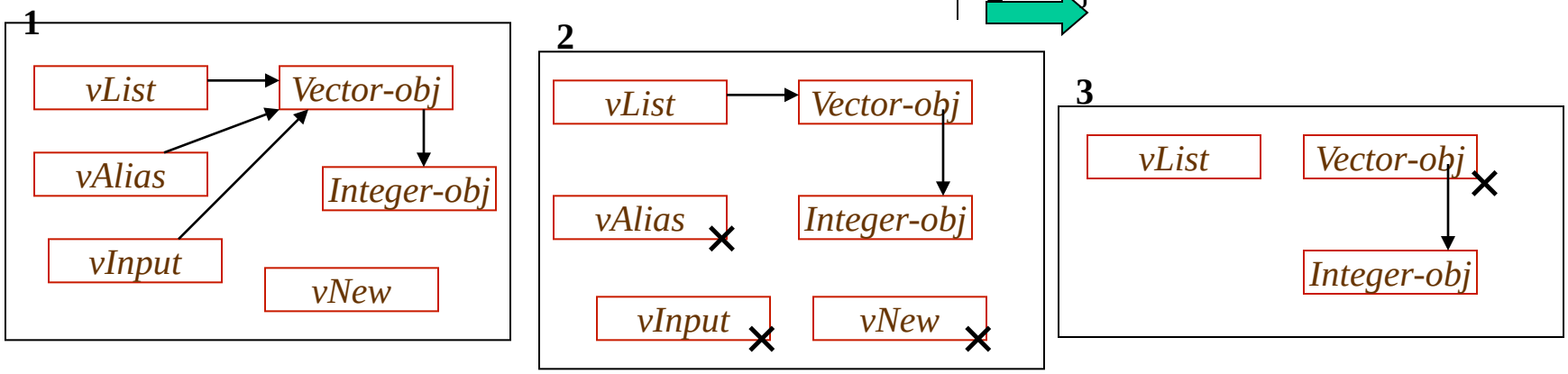


Pointers cont'd

- **GC Algorithms:**
  - Mark and sweep GC:
    - Sweep through entire memory looking for referenced blocks and free unused blocks
    - Problems:
      - multi pass processing causes delay in execution of programs
    - Example:

```
Vector vList;
vList = new Vector( );
vList.addElement( new Integer(1));
calc( vList );
……..
vList = null;
```

```
public void calc(Vector vInput)
{
  Vector vNew,vAlias;

  vNew = new Vector();
  vAlias = vInput;
  ….
}
```



Pointers cont'd

- **GC Algorithms:**
  - Languages with GC are unsuitable for real time programming….

```
public class foo
{
  public foo()
  {
    StringBuffer strbufLocal;
    strbufLocal = new StringBuffer( );

    for(int nIndex = 1; nIndex < 20000; nIndex++)
    {
    StringBuffer strbufTemp = new StringBuffer();
    strbufTemp.append("x");
    strbufLocal.append( strbufTemp );
    int nLength = strbufTemp.length();
    }

  }
  public static void main(String strarrArgs[])
  {
    foo f = new foo( );
  }
}
```

```
C:\>java -verbose:gc foo
[GC 511K->182K(1984K), 0.0208331 secs]
[GC 694K->196K(1984K), 0.0045478 secs]
[GC 708K->214K(1984K), 0.0026766 secs]

C:\>java -verbose:gc foo
[GC 511K->182K(1984K), 0.0209680 secs]
[GC 694K->196K(1984K), 0.0046308 secs]
[GC 708K->214K(1984K), 0.0026199 secs]

C:\>java -verbose:gc foo
[GC 511K->182K(1984K), 0.0206537 secs]
[GC 694K->196K(1984K), 0.0045754 secs]
[GC 708K->214K(1984K), 0.0026168 secs]

C:\>
```

Data types Summary

- **Primitive types**
  - **Integer, Float, Boolean, Char, Pointers**
- **Structured Types**
  - **Strings, Ordinal, Arrays, Associative Arrays, Records, Union, Lists**
- **Object Type**
- **Class Type**
- **Function Type**

Object Type

- **The language would have means to create an instance from an encapsulated structure that has functions + attributes**

- **Object-Based languages refers to having objects without classes and classical inheritance**
  - E.g. Ada 83, Modula-2, Javascript

Object Type

- **Javascript:**

```
// creating our own object
personObj=new Object();
personObj.firstname="John";
personObj.lastname="Doe";
personObj.age=50;
personObj.eyecolor="blue";
```

Object Type

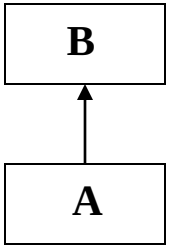
- **Javascript:**

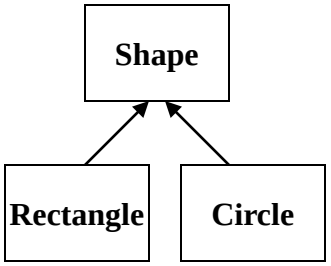
```
function person(firstname,lastname,age,eyecolor)
{
        this.firstname=firstname;
        this.lastname=lastname;
        this.age=age;
        this.eyecolor=eyecolor;
}

var myFather   = new person("John","Doe",50,"blue");
var myMother = new person("Sally","Rally",48,"green");
```

Class Type

- **Classes describe the rules by which objects behave; those objects, described by a particular class, are known as "instances" of said class**

- **Subtypes**
  - If given type A is compatible with type B, then A is a subtype of B
  - Hence, one datatype can be more than one subtype
  - Polymorphism
    - E.g.





Class Type

- **Interfaces/protocols**
  - A definition of methods and values which the objects agree upon in order to cooperate.
  - A specification of those properties of a software component that other components may rely upon
  - E.g.

```
public interface Shape{
        public abstract void draw( int x, int y);
}
```

Function Type

- **Function types**
  - A type that allow an object to be invoked or called as if it were an ordinary function

```
// Declaration of C sorting function
void sort (int [] itemlist, int numitems,  int (*cmpfunc)(int*, int*)  );
...

// Callback function
int compare_function( item* A, item* B)
{
  //……
  //……
}

sort( itemlist, numitems, compare_function);
```

Data types Summary

- **Primitive types**
  - **Integer, Float, Boolean, Char**
- **Structured Types**
  - **Strings, Ordinal, Arrays, Associative Arrays, Records, Union, Lists**
- **Object Type**
- **Class Type**
- **Function Type**

Note: Don't confuse data types with API data structures (BST, Graphs,…)