Storage

Outline

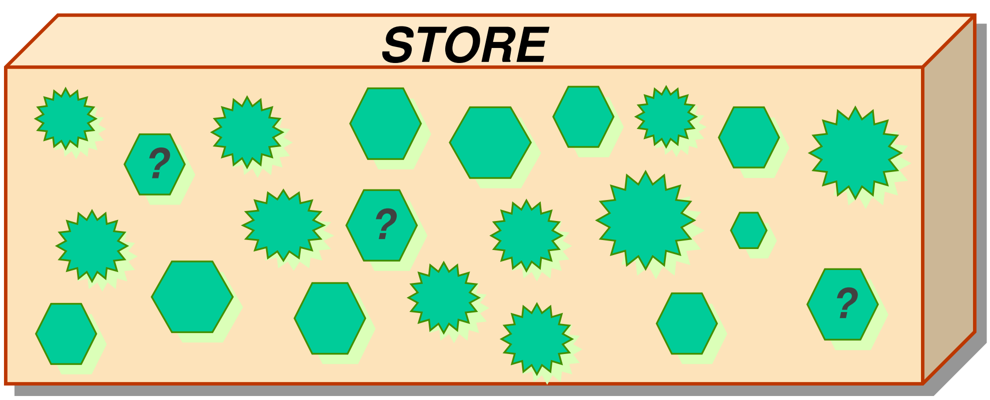
- Variables and updating
- Composite variables
  - **Total and selective updating**
  - **Array variables**
- Storables
- Copy *vs.* Ref semantics
- Lifetime
  - **Local and global variables**
  - **Heap variables**
  - **Persistent variables**
- Dangling References and Garbage collection

Variables and Updating

- **Variable**: an entity that contains a *value*.
  - **Values may be primitive or composite**
  - **There are two basic operations that can be performed on a variable:**
    - Inspect
    - Update
- **Model real-world objects *with state*:** e.g. a country's population
- Variables in imperative languages *vs.* mathematical variables:
  - **Mathematical variables stand for *fixed* but unknown values**
  - **Variables in imperative languages may *change* over time (`n := n+1`)**
  - **They don't necessarily have a value at all**
- Variables may be:
  - **Short-lived** - created and used within one program, or
  - **Long-lived (persistent)** – exist independently of programs: files and databases.
- Variable are realized by a *storage medium* (memory, disk, etc.).

A Model of Storage

- *Store*: a collection of *cells*
- **Cells**: *allocated* or *unallocated*.
- An allocated cell has *content*: a *storable value* or *undefined*.



Example:

```
var n: Integer;       { Some unallocated cell changes status to allocated }
begin
    n := 1            { Content changes from undefined to 1 }
    n := n + 1        { Content changes from 1 to 2          }
end;                  { Cells changes status to unallocated   }
```

Composite Variables

- **Composite value:** Has subcomponent values, which may be *inspected* selectively.
- **Composite variable:** Has subcomponent variables. These may be *inspected* and (sometimes) *updated* separately.
  - It is always possible to make selective *inspection*, since once the *value* in a variable is inspected, you can selectively inspect each component.

The Structure of Composite Variables

- **In most languages:** a ***variable*** of type *T* is structured like a ***value*** of type *T*
  - **Exception:** Packed arrays in Pascal, which cannot be accessed before the array is unpacked
  - Consider `bignum`s
- A record variable is a tuple of variables:
  - **`type`** `Date =` **`record`** `m: Month; d: 1..31` **`end;`**
  - **`var`** `today: Date;`

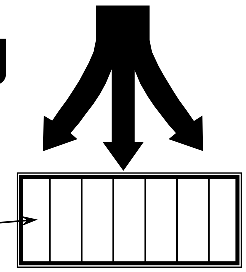
today

today.m	today.d

- An array variable is a mapping into variables:
  - **`var`** `holidays:` **`array`**`[1..30]` **`of`** `Date;`

Other type constructors?
Sets? Unions?

Total and Selective Updating



- Total updating (and selective inspection):
  - `today := holidays[1];`
- Selective updating:
  - `holidays[1] := today;`
  - Can be viewed as total updating that happens to leave some components unchanged.
- Selective updating is not essential:
  - **ML variables:** *T* `ref` - a variable which can store a value of type *T.*
  - **If *T* is a record then no selective updating is possible.**

```
datatype month = jan | feb | mar | ... | dec
 type date = month * int
 val today: date ref = ref(feb, 23)
 ...
 today := (feb, 29)
```

This is an aggregate!

  - **If the components are a reference type, then the whole cannot be updated.**
  - **ML arrays - only selective updating is possible:** All array components are references.

Array Variables

- An array is a mapping from an *index set* to a *collection of variables*.
- When is the index set determined?
  - **Static arrays:** fixed at compile time.
  - **Dynamic arrays:** on creation of the array variable.
    In Ada:

```
m: Integer := ...;
...
type Vector is array (Integer range <>) of Float;
a: Vector(1..10);
b: Vector(0..m)
procedure ReadVec(v: out Vector) is ...;-- Use v'first & v'last
ReadVec(a); ReadVec(b)
...
a := b; -- Succeeds only if b has exactly 10 elements.
```

  - **Flexible arrays:** not fixed; bounds may change whenever index is changed.
- The type of these arrays (flexible or dynamic) is:
  - **Integer × Integer × (Integer → Real)**

Generalized Arrays

- **Ordinary arrays**: mappings from integral types.
  - **Advantages of integral indices:**
    - Only values are stored, not indices.
    - The bounds of the array define its legal indices.
  - **Disadvantages:**
    - When data are sparse, packing techniques are needed.
    - Inflexible programming.
- **Generalized (associative) arrays:** mappings from non-integral types.
- **Example:**

```
Element "dog" Value "chien"
Element "cat" Value "chat"
Element "one" Value "un"
Element 1     Value "un"
```


Generalized Arrays in AWK

- In AWK any string can serve as an index of an array:
  - Built-in constructs to loop over all valid indices of an array
  - Built-in constructs to check if a value is a valid index of an array
- **Type**: Since for any potential string we can *determine* whether it is a valid index and *access* the value at that index, the type of an array of strings indexed by strings is: $\wp$(*String*) X (*String* → *String*)

  An array in AWK can be viewed as a pair `<S,f>`, where `S` is a set of strings and `f` is a function from strings to strings.
- In AWK, if you try to access an undefined index, an element with this index is automatically created and updated with the null string. Then the null string is returned.
- **Note:** this is different from having only a mapping `f`:*String* → *String,* which does not distinguish indices not in the array from indices whose cell is empty.
- Thus, AWK defines for every array `<S,f>`:
  - `f` maps every string not in `S` to the empty string
  - An expression like `<S,f>[x]` (accessing the cell with index `x`) has the value `f(x)`, with the side effect $S := S \cup \{x\}$

Storables

- **Storables:** values which can be stored in a *single cell* and cannot be *selectively* updated
  - ***The "atoms" of storage***
- Pascal storables:
  - **primitive types**
  - **sets**: cannot be selectively updated
  - **pointers**
- Pascal non-storables:
  - **Arrays, records and files:** can be selectively updated
  - **Procedure and functions:** cannot be stored
  - **References:** cannot be stored
- ML storables: everything except arrays
  - **primitive values**
  - **records, constructions and lists**
  - **functions**
  - **references**

More on Storables

- The primitive values and pointers are storables in most programming languages
  - **As usual, strings may be an exception**
- Simple variables are those with values that are storables
- What about functions?
  - **Sometimes Not storable at all….no assignment in some languages**
  - **Functional: Can be seen as a storable, since it has no components**

Reference *vs.* Value Semantics

- Copying and comparison of pointers have potentially two semantics:
  - **Value semantics:** the actual values are copied and compared
    - **Deep semantics:** the whole network of objects which can be accessed are copied and compared
    - **Shallow semantics:** only the first level is copied and compared
  - **Reference semantics:** only the references are copied
- Comparison:
  - **Value semantics:** slow, defines ownership relationship between objects, requires run-time type information stored with each composite object (for deep value semantics)
  - **Reference semantics:** fast, objects could be shared, allows values which are more complicated than what can be captured by algebraic types, no need to store type with objects

Value *vs.* Reference Semantics (cont.'d)

- Value *vs.* reference semantics in contemporary languages:
  - **Value semantics languages:** Lisp, ML, Prolog
  - **Reference semantics languages:** Java, Smalltalk
  - **Mixed semantics languages:** Eiffel, C++
  - **Most languages have some kind of a mix:**
    - In Java, primitive types have value semantics
    - There are hacks in Lisp that allow reference semantics
    - References in ML allow reference semantics
- **Lazy copying:** an implementation technique of value semantics, where a copy of a large object is made by using a reference. The actual value copy operation is made when (and if) the source or the destination variables are modified

Copy (Value) *vs.* Reference Semantics

- Even assignment (x := v) can be tricky!
- For primitive values, a copy of v is put in x (no problem).
- What happens when a composite value (from a variable) is assigned to a variable of the same type? (x := v)
- Copy (or Value) semantics: all components of the composite value are copied to the corresponding components of the variable
  - **C, C++, Ada**
- Reference semantics: the composite variable will have a pointer or reference to the composite value
  - **Java**
- In copy semantics, later changes in parts of x have no effect on v, while in reference semantics, they affect all references to v.

C++ *vs.* Java

- C++ copy semantics

```
    struct Date { int y, m, d;  };
Date dateA = {2006, 11, 27};
Date dateB ;
dateB = dateA
```

- Java reference semantics

```
    class date { int y, m, d;
        public Date (int y, int m, int d) {…}
     }
Date dateR = new Date(2007, 12, 15);
Date dateS = new Date(2007, 11, 1);
dateS = dateR
```

- dateA.m = 12   has no effect on dateB in the C++ example
- dateS.m = 12 changes dateR in the Java example!

Workarounds

- For C++ (normally copy/value semantics) use pointers:

```
Date* dateP = new Date;
Date* dateQ = new Date;
*dateP = dateA;
 dateQ = dateP;
```

Now dateP and dateQ point to the same variable. Selective update of *dateP will also change *dateQ and vice versa…

- ML has value semantics, but can use ref's to get reference sem.
- For Java (normally reference semantics) use "clone':

```
Date dateT = new Date(2006, 1, 1);
dateT = dateR.clone();
```

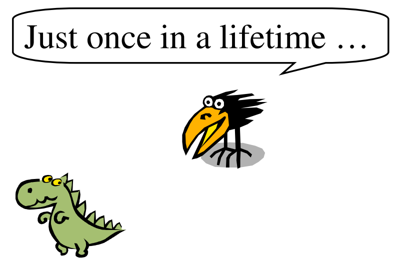
Now dateT has a new copy of the value of dateR. A common error in Java: forgetting to use clone when copy semantics is needed

Equality Testing

- Always consistent with the semantics of assignment
- Copy semantics: compare the components, one by one
  - **Can be expensive**
  - **Requires type information for each component**
- Reference semantics: are the pointers equal?
  - **Very inexpensive**
  - **No need to carry around type information during runtime, except if selective updating is possible**
- Always allows concluding that the values of x and y are equal after x:= y, no matter which semantics is used.

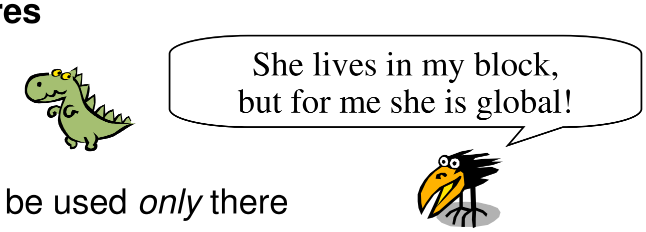
Lifetime

- The interval between the ***creation*** (allocation) and the ***deletion*** (deallocation) of a variable is called its ***lifetime***
- Lifetime management is important for economic usage of memory
- Kinds of lifetime:
  - <u>Block activation</u>**: local** variables
  - <u>Programmer's decision</u>**: global** variables
  - <u>Program activation</u>**: heap** variables
  - <u>Permanent</u>: **persistent** variables



Local and Global Variables

- **Block:**
  - **Pascal functions and procedures**
  - **ML's *let* expressions**
  - **C's { }**
- **Block variables:**
  - **Local**: declared within a block to be used *only* there
    - Usually, to make the compiler's job easier, declarations are made at the beginning of the block, but not always. In C++ declarations can be made anywhere in a block
  - **Global**: a local variable declared in the outermost block
- **Block activation:**
  - The time interval during which the block is executed
  - One local variable *name* may stand in fact for different variables:
    - Separate definitions in disjoint blocks
    - Several lifetimes - when the block containing it is activated several times
    - Multiple lives - when the block is furthermore activated recursively

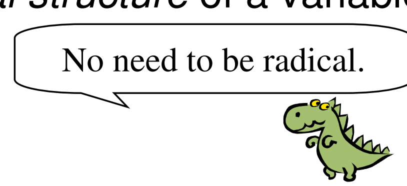


Static Variables

- **C and PL/I:** *static* variables – variables whose lifetime is the whole program execution, independently of their declaration's location
- The goal is to allow to store values that are needed in different activations of the same block/module, regardless of the block structure
- However, this goal is better achieved with OOP

Heap Variables

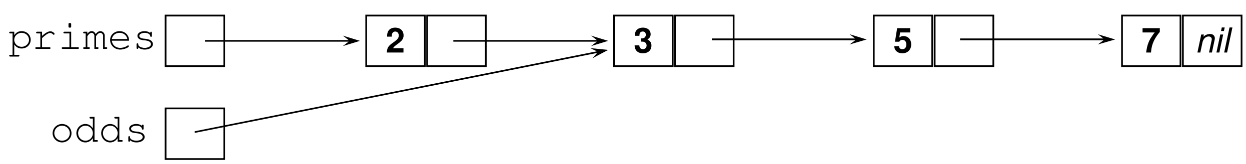
- Can be created and deleted *at will*, using the operations:
  - **allocate**
  - **deallocate**
- Anonymous
- Accessed by **pointers:**
  - Pointers are first class values
  - Allow arbitrary directed graphs
  - Allow modifications that are more ***radical*** than selective updating, modifying the *actual structure* of a variable



Using Heap Variables

```
type  IntList = ^ IntNode;
      IntNode = record head: Integer; tail: IntList; end;

var   odds, primes: IntList;

function cons (h: Integer; t: IntList): IntList
var l: IntList
begin
      new(l);
      l^.head := h; l^.tail := t;
      cons := l;
end;

...

odds   := cons(3, cons(5, cons(7, nil)));
primes := cons(2, odds);
```

A pointer value is either nil or an address of a variable



Pointers + Heap Variables = Recursive Types

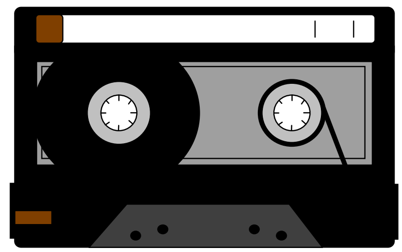
- Pointers and heap variables are error-prone:
  - **What's** `p^.tail := q`?
    - *Which* list is selectively updated?
    - Is it only *one* list which is updated?
    - Did it change a data structure by introducing a *cycle*?
  - **Must use discipline, care and debuggers**
- Suppose that Pascal had list type:

```
var primes, odds: list of Integer;
```

  - What is the meaning of: `primes := odds`?
    - Reference copying:
      - **Inconsistent with arrays, records and primitive types**
      - **Pointers in disguise**
      - **Selective updates to one will affect the other**
    - Data copying:
      - **Inefficient, but natural**
    - Possible solutions: *prohibit* selective update (Lisp), or *lazy* copying

Persistent Variables

- **Lifetime:**
  - **Transient:** lifetime bounded by the activation of the program that created it.
    - Local, Global, Heap variables
    - Other file variables
  - **Persistent:** lifetime transcends an activation of any particular program.
    - In Pascal persistent variables are *program parameters*, which cannot be deleted or created by the program
    - In C and Ada there are standard I/O libraries/packages that deal with files
- **Files** are composite variables:
  - **Serial file:** a sequence of components
  - **Direct file:** an array of components
- Access is restricted for efficiency reasons. Pascal examples:
  - **Inspect:** sequential reading
  - **Update:** emptying and sequential write
  - **Assignment:** forbidden



Using Files as Persistent Variables (Pascal)

```
program POP (st_file);
type Country = (DK,GB,...,NL);
     Statistics = record
       population: 0..100000000;
     end;
var stats : array [Country] of Statistics;
    st_file : file of Statistics;
procedure readstats;
     var cy : Country;
     begin
       reset(st_file);
       for cy:=DK to NL do read(st_file, stats[cy]);
end;
begin
     readstats;
     ...  stats[cy].population ...
end.
```

Inherent Persistent Variables (hypothetical...)

```
program POP(stats);
type Country = (DK,GB,...,NL);
     Statistics = record
        population: 0..100000000;
     end;
var stats : array [Country] of Statistics;
     (* Removed: st_file: file of Statistics; *)
     cy : Country;
(* Removed:
procedure readstats;
     var cy : Country;
     begin
        reset(st_file);
        for cy:=DK to NL do read(st_file, stats[cy]);
end; *)
begin
     (* Removed: readstats; *)
     ...  stats[cy].population ...
end.
```


Reflection on Persistent Variables

- Similar to variables:
  - Arbitrary lifetime
  - Nested lifetime on some systems
  - File store is just like the ordinary store model
- Unlike variables (in Pascal):
  - Transient variables can be of any type, including `file`
  - Persistent variables must be of type `file`
- Type completeness principle requires persistent variables of all types. For instance:
  - Persistent variables of primitive types
  - Persistent array variables = direct files

If files were like ordinary variables, then all I/O programming work would be saved.

Dangling References

- If we deallocate a heap variable, there might still be pointers to it. These are dangling references….if we try to use them very strange things might happen
- Therefore deallocation is considered dangerous…
- The alternative: garbage collection (will discuss in a minute)

- If we can still refer to a local variable after the relevant block is finished, we also get dangling references

Dangling References

- Suppose that Pascal was like C...

```
var r: ^Integer;

procedure P;
var v: Integer;
begin r := &v end;

begin
  P;
  r ^:= 1   ☹
end;
```

- **Dangling Reference:** an attempt to access a variable which is no longer alive
- Pascal prevents this type of dangling reference

Function Variables and Dangling References

Suppose that Pascal had function variables...

```
var fv: Integer -> Char;

   procedure P;
   var v: ...;
      function f(n: Integer): Char;
      begin
         ...v...
      end;
   begin
      fv := &f
   end

begin
   P;
   ... fv(0) ...   ☹
end;
```

This is one of the reasons why C doesn't have nested functions

More on Dangling References

- **Another form of the previous problem occurs with a function returning a procedure variable**
- **Above problems can be solved by:**
  - **Algol-68:** a reference to a local variable cannot be assigned to a variable with a longer lifetime:
    - Run time check
    - Awkward restrictions on programmer

    Can you explain why?
- **Heap variables**: reference to a variable which was deallocated:
  - **Elimination of dispose**
  - **Garbage collection**
- **The ultimate solution:** treat all variables as heap variables:
  - **All functional languages, Prolog**
  - **Inefficient**

Garbage Collection

- With pointers, memory management becomes quite complicated, and errors of two kinds often occur:
  - **Dangling references:** access to variables which are no longer alive
  - **Memory leaks:** unused memory that is not deallocated
- **Garbage collection:** automatic management of memory
  - **User never deallocates memory**
  - **When memory becomes scarce, a garbage collection procedure is applied to collect all unused memory locations**
  - **Mark and sweep:** the simplest mechanism for collecting unused memory cells
    - **Mark**: mark all cells as unused
    - **Sweep**: unmark all cells in use (stack, global variables), and cells which can be accessed from these
    - **Release:** all cells which remain marked

Memory Leaks and Garbage Collection?

- **Memory leaks:** usually happen in a non-garbage collecting system, when a programmer forgets to deallocate memory
- Garbage collecting systems rely on the programmer to nullify pointers so that the system would know that they are unused
- If the programmer forgets to assign `nil` to pointers, then the memory is held even though it is not used
- Common problem in Java programming!

Inefficiency of Garbage Collection

- In a garbage collecting system, there are no stack variables. It is unsafe to deallocate automatic variables
- The following code may slow the Java language processor (which provides garbage collecting) down to a halt, whereas it would be a snap in C/C++

```
void f()
{
  data a[10000];
}

...

for (i = 0; i < 10000; i++)
  f();
```

- **Escape analysis:** an active area of research in Java. Try to determine for each automatic variable if it can "escape" a function, and if not, treat it like a stack variable

Summary

- Variables and storage models can differ among languages
- Updating can be total or selective for composite variables
- Assigning values to variables has different possible semantics when the values are composite
  - **Copy/ Value semantics**
  - **Reference semantics**
- Variables have a lifetime (global, local, heap, persistent)
- Dangling references can arise from
  - **Deallocating heap variables**
  - **Using pointers to define references to local variables that are then used outside the local block**
  - **Activating a function outside the enclosing block where it is defined when it uses variables local to that block**
- Garbage collection is often used instead of deallocation