Section 5

Storage

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing

5. Storage
   5.1 Storage models
   5.2 Arrays
   5.3 Variables’ life time
   5.4 Representation of types in memory
   5.5 Automatic memory management
   5.6 Run time type information
5. Storage

5.1. Storage models
Storage models: a visual mindmap
Storage & the need for variables

**Functional Paradigm**
- No variables.
- Values which might be named.

**Logic Programming Paradigm**
- No variables
- Values which might be named
- Math-like “variables”, denoting something which is yet undetermined
- Once determined, value does not change

**Imperative Paradigm**
- Variables denote values that might change
- Closer to the machine
- Useful for modeling real life quantities, e.g., person’s weight
Variables

**Definition (Variable)**

An entity that may contain a value and provides inspect and update operations on its content.

- Realized by *storage* medium, e.g.,
  - memory
  - disk
- Very different from mathematical “variables”
  - Mathematical variables: fixed but possibly unknown values
  - Logic programming variables: just like mathematical variables
  - Imperative variables: may change over time: \( n := n+1; \)
  - may not have a value at all
5. Storage

5.1. Storage models

5.1.1. Utopic perspective of memory
Store, cells & values

*Store* is another name for memory, which has unboundedly many “*cells*”. Some of which are “*persistent*”, while others are “*ephemeral*”. Some cells are “*free*”; others are “*allocated*”. Allocated cells could be “*uninitialized*”; initialized cells may contain *integers*, or *strings*, or any *arbitrary value*. Storage = collection of cells which may be in a variety of states.

![Diagram of cells with various values and states]
Confusing terminology: variables vs. cells?

- Strictly speaking, an *allocated* cell is the “implementation” of a variable.
- But, the terms are often used interchangeably.
- Usually, when one says “variable”, one means a *named variable*.
- But, there are anonymous variables.
- Or perhaps, we should reserved the term cell for “anonymous variables”.
- Do whatever you please; in this course, the terms are synonymous.
5. Storage

5.1. Storage models / 5.1.1. Utopic perspective of memory

Life cycle of a cell in the store

Our cell, just as all other cells is born **free**

Let’s define a C++ function, and, at the same time, execute it

The function **allocates** a variable

And then **initializes** it

It then **inspects** the variable

The inspection of an uninitialized variable is **undefined**

Our function then **updates** the variable

and inspects it...

The variable is deallocated when the function returns

```cpp
goo()
{
    int n;
    n = 6;
    cout << n;
    n = 28;
    cout << n;
    cout << n;
}  
```
Cells & references to cells in ML

- allocate a cell of type `int`; store `3` in it; and, let `r` be a reference to this cell, i.e., `r` is a name of the value of the reference to this cell:
  ```ml
  val r = ref 3;
  val r = ref 3 : int ref
  ```

- let `s` be another name for the value representing a reference to the same cell:
  ```ml
  val s = r;
  val s = ref 3 : int ref
  ```

`r` and `s` are not cells; they are merely names of values.
Retrieving cell contents in ML

- **get 3**, the contents of this cell:

  ```ml
  !r;
  val it = 3 : int
  ```

- **!s** is the same as **!r**:

  ```ml
  !s;
  val it = 3 : int
  ```

- use cell’s contents to create a new named value:

  ```ml
  val x = !r + 2;
  val x = 5 : int
  ```

**New name of a value**

The newly created name, `x`, just as `r` and `s`, is not a cell; it is a name of a value.
Mutable cells in ML

The ML memory model is close to the utopic

- change the value stored in this cell:
  
  ```ml
  r := 2;
  val it = () : unit
  ```

- get this value:
  
  ```ml
  !r;
  val it = 2 : int
  ```

- get the same value:
  
  ```ml
  !s;
  val it = 2 : int
  ```
More on mutable cells in ML

- change the contents of this cell:

\[
\begin{align*}
\text{s} & := \text{!r + 4}; \\
\text{val it} & = () : \text{unit}
\end{align*}
\]

- now get the content of the cell referenced by \text{r}:

\[
\begin{align*}
\text{!r}; \\
\text{val it} & = 6 : \text{int}
\end{align*}
\]
Left vs. right occurrence of variable

What’s the difference between the two occurrences of \( v \)?

Left vs. right

\[ v \leftarrow \varphi(v, e_1, e_2, \ldots, e_n) \]

Left hand side

\[ a[3*a[i*2] - 2*a[i*3]] := 0 \]

1. Evaluate \( v \) (even in a “very basic” PL, \( v \) may be the result of an expression)
2. Treat the result as reference to a cell
3. Use this reference as the value of \( v \)
4. Get ready to assign something to that cell

Right hand side

\[ t := a[3*a[i*2] - 2*a[i*3]] \]

1. Retrieve the contents of that cell
2. Use this contents as the value of \( v \)
3. You can forget about the cell now
L-Value vs. R-value of an expression

C, C++, Pascal, and most other PLs make a distinction between the *L-value* and the *R-value* of an expression.

- All expressions have an R-value.
- Only particular expressions have an L-value.
- The distinction between the two is determined by context.

C, and more so C++, has fairly sophisticated L-values.

```c
while (*s++ = *t++);
```

The expression *s++ has an L-value!

```cpp
int& min(int &x, int &y) {
    return x < y ? x : y;
}
min(a, b) = min(c, d);
```

Function `min` returns an expression which has an L-value (just as an R-value).
L-values vs. R-values in ML

Distinction between L-values and R-values in ML is simple:

If \( a \) is a name of a value, created by

```ml
val a = 19;
```

then:

- \( a \) does not designate a memory cell
- \( a \) is just name of a value
- \( a \) cannot be changed
- \( a \) can only
  - go out of scope (in an enclosing context)
  - be hidden (in an enclosed context)

If \( r \) is a reference to a cell, created by

```ml
val r = ref 17;
```

then:

- \( !r \) is the contents of this cell (R-value); it can be used e.g., by
  ```ml
  val x = !r;
  ```
- \( r \) is the reference to this cell (L-value); it can be used e.g., by
  ```ml
  r := x + 3;
  ```
More on L-values in C

Not every value is an L-value:

```c
0 = 1; // ✗ not an L-value
```

Not every L-value is modifiable:

```c
const int i = 0;
i = 1; // ✗ unmodifiable L-value
```

There is an implicit conversion from L-value to R-value:

```c
int max(int & x, int & y) {
    return x + y - min(x,y); // ✓ implicit conversion
}
max(a, b) = max(c, d); // ✗ not an L-value
```

C’s address taking operator, &, is applicable to, and only to, L-values:

```c
&1; // ✗ not an L-value
&i; // ✓ an (unmodifiable) L-value
&max(a,b); // ✗ not an L-value
&min(a,b); // ✓ is an L-value!
```
Composite variables

Normally, a variable of type $T$ is structured like a value of type $T$

*Oddballs exist, e.g., packed arrays in Pascal, which cannot be accessed before the array is unpacked*

A record variable is a tuple of variables:

```pascal
TYPE
  Date = Record of
  m: Month;
  d: 1..31;
  y: Integer
end;

VAR
  today: Date;
```

- `today` access the entire value stored in this variable
- `today.d` access a component of the value stored this variable
Total/selective inspect/update

**Composite value**  Has subcomponent values, which may be inspected selectively.

**Composite variable**  Has subcomponent variables. These may be inspected and (sometimes) updated separately.
Total/selective inspect/update in ML

Define a reference to a composite cell

```ml
val v = ref {first="Yogi", last="Bear"};
```

**Total inspect**

```ml
!v;
val it = {first="Yogi", last="Bear"} : {first:string, last:string}
```

**Total update**

```ml
v := {first="Yogi", last="Berra"};
!v;
val it = () : unit
val it = first="Yogi",last="Berra" : {first:string, last:string}
```

**Selective inspect**

```ml
(#first)={!v);
val it = "Yogi": string
```

**Selective update??**

```ml
(#last)={!v} := "Berra";
Error: operator and operand don't agree [tycon mismatch]...
```
Ban on selective update

- It is always possible to make selective inspection, since once the value in a variable is inspected, you can selectively inspect each component.
- Normally, selective update is also possible (update a single field from a record).
- In some cases, in some languages, only total updates are possible (update all fields, or none).
Total and selective update in C

Composite variables can be inspected and updated in total or selectively

```c
struct Complex {
    double x, y;
} a, b;
...

a = b; // Total update (and total inspect)
double z = b.y * a.x; // Selective inspections
a.x = z // Selective update
```

Atomic variable single cell
Composite variable nested cells
5. Storage

5.1. Storage models

5.1.2. Real-world memory models
Memory structure: the 8086 architecture

16-bit Segment Register

CS, DS, SS, ES

16-bit Offset Register

AX, BX, CX, DX, SP, BP, SI, DI

20 Bits Address

Segment Registers

<table>
<thead>
<tr>
<th>DS</th>
<th>Data Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Code Segment</td>
</tr>
<tr>
<td>SS</td>
<td>Stack Segment</td>
</tr>
<tr>
<td>ES</td>
<td>Extra Segment</td>
</tr>
</tbody>
</table>

Offset Registers

<table>
<thead>
<tr>
<th>AX, BX, CX, DX</th>
<th>General Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>BP</td>
<td>Back Pointer</td>
</tr>
<tr>
<td>SI, DI</td>
<td>Offset Registers</td>
</tr>
</tbody>
</table>
Extended vs. expanded memory in the ancient 8086 architecture

- **Conventional Memory.** Accessible to software.
- **Upper Memory Area.** Accessible to software, but reserved for screen and other I/O memory map.
- **High Memory Area.** Accessible to software, but only on certain architecture variants.
- **Extended Memory.** To access, need to switch to protected mode, copy to UMA, and revert to real mode.
- **Expanded Memory.** Early, less-elegant, but more popular version of extended memory. (not shown on diagram)
5. Storage

5.1. Storage models

5.1.3. Classical storage model
Classical model of memory

Segments:

- **Zero**
- **Code**
- **Constants**
- **Data**
- **Heap**
- **Stack**

Permissions:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Code</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Constants</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Data</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Heap</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Unallocated</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Stack</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
C programs & the classical memory model

Where does each identifier in the following program reside in the classical memory map? Are there any identifiers which are not mapped to memory? The programs has two nameless entities which are still found in the above map. Which are they? Where do they reside?

```c
#include <stdio.h>
#include <stdlib.h>

long fib(int n) {
    static int N;
    auto long r = (n <= 1? 1 : fib(n-1) + fib(n-2));
    printf("Call\#%d\n",++N);
    return r;
}

enum { N = 20 };
long *r;

int main() {
    int i;
    r = malloc(N * sizeof(long));
    for (i = 0; i < N; i++)
        r[i] = fib(i);
    return r[N-1] + r[N-3];
}
```
5. Storage

5.2. Arrays

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing

5. Storage

5.1. Storage models
5.2. Arrays

5.2.1. Varieties of arrays
5.2.2. Arrays with integral index types
5.2.3. Type of arrays
5.3. Variables’ life time
5.4. Representation of types in memory
5.5. Automatic memory management
5.6. Run time type information
Array variables

```
VAR
  holidays: Array[1..30] of Date;
```

**Definition (Array values)**

An *array value* is a mapping from a set of indices to a set of values.

**Definition (Array variables)**

An *array variable* is a realization of array value using variables, so that each of the image of each index may be changed at runtime.
Why only now?

- Array values are not very useful (Did you see any arrays in ML?)
- But... array variables become very useful...
  - Efficient mapping into memory with the classical storage models
  - Foundation for many algorithms
  - Foundation for many data structures
5. Storage

5.2. Arrays

5.2.1. Varieties of arrays
Variety I/V: static arrays

- size determined at compile time
- allocated on the data segment

```c
const char* days[] = {
    "Sun", "Mon", "Tue",
    "Wed", "Thu", "Fri", "Sat"
}; // An array literal
int main(...) {...}
```
Variety II/V: Stack based arrays

- Size determined at runtime
- Size but cannot change after creation
- Allocated on the stack
- The only kind of arrays in Pascal
- Required that index was a compile-time constant in early versions of C
- Added, after noticing that they do not violate the no hidden costs principle:
  - Creation is by mere subtraction of a value from the stack pointer
  - Time to create is $O(1)$
  - Size can be negative, but C programmers are accountable and responsible.

```c
void fileCopy(FILE *from, FILE *to) {
    char buffer[1 << 12]
    ...
}
```

```c
void printPrimes(int n) {
    unsigned char sieve[n];
    ...
}
```
Variety III/V: Dynamic arrays

```c
int[] printPrimes(int n) {
    unsigned char sieve[n];
    ...
    int r[] = malloc(sum(sieve) * sizeof(int));
    ...
    return r;
}
```

- size determined at runtime
- size cannot change after creation
- allocated on the heap segment
Variety IV/V: flexible arrays

- size may change at runtime
- size may change after creation
- array may expand or shrink
- found e.g., in Perl

```perl
@a = 1..6;  # uninitialized; size 6
@a = (1,2,3);  # initialized; size 3
@a[13] = 17;  # size is now 13
@a[17] = 13;  # size is now 17
delete @a[17];  # size is now 13
delete @a[13];  # size is now 3
```
Variety V/V: associative arrays

$wives["Adam"] = "Eve";
$wives["Lamech"] = "Adah and Zillah";
$wives["Abraham"] = "Sarah";
$wives["Isaac"] = "Rebecca";
$wives["Jacob"] = "Leah and Rachel";
...
...
echo $patriarch;
echo $wives[$patriarch];

- index can be anything, typically strings.
- common in scripting PLs, e.g., AWK, JavaScript, PHP
- typically, implemented as a hash table
The unbelievable power of associative arrays

Using AWK to compute the frequency of words in the input stream:

```
#!/usr/bin/awk -f
{ for (i = 1; i <= NF; i++) a[$i]++;}
END {
  for (w in a)
    if (a[w] in b)
      b[a[w]] = b[a[w]] "", w;
    else {
      b[a[w]] = w;
      if (max < a[w]) max = a[w]
    }
  for (; max > 0; max--)
    if (b[max] != "")
      print max, b[max];
}
```

Explanation follows...
### Computing word frequencies in AWK

AWK’s implicit loop reads lines in turn, breaking each line to space-separated “fields”.

```
#!/usr/bin/awk -f

# implicitly executed
# for each input line
{
    for (i = 1; i <= NF; i++)
        a[$i]++;
        # optional semicolon (;)
    # The "$" character is special:
    # variable $i is the i\textsuperscript{th}
    # word in the current line
}

END {
    # after last line read
    # Accumulate in b[i] all words
    # that occur i times
    max = 0;  # not really necessary
    for (w in a) {
        if (! (a[w] in b)) {
            b[a[w]] = w
            if (max < a[w])
                max = a[w]
        } else
            b[a[w]] = b[a[w]] "\n" w;
    } else
        b[a[w]] = b[a[w]] "\n" w;
    # Print array b in descending order
    for (; max > 0; max--)
        if (b[max] != ")
            print max, b[max];
}
```
Summary: determining the index set

When is the index set determined?

**Static Arrays**  fixed at compile time.

**Dynamic Arrays**  on creation of the array variable.

**Stack Based Arrays**  on creation of the array variable.

**Flexible arrays**  not fixed; bounds may change whenever index is changed.

**Associative Arrays**  no “bounds” for the set of indices; the set changes dynamically as entries are added or removed from the array.
Arrays’ efficiency

*Static, Stack based, and Dynamic:* efficient implementation in the classical memory model.

- including range-based arrays, as in Pascal
- including true multidimensional arrays, as in Fortran
- including arrays of arrays, as in C

*Flexible and Associative:* require more sophisticated data structure to map to the classical memory model.
Sophisticated data structures as part of PLs?

- Associative arrays are great!
- We want more, ...
  - sets!
  - multisets!!
  - stacks and queues and trees!!!
The sad story of Pascal’s sets

- simple implementation
- efficient implementation
- does not scale
- with scale, you need to carefully balance
  - operations repertoire
  - time
  - memory
  - parallelism
Dilemmas in language design

- Which, if any, sophisticated data structures should be part of the PL?
- Which, if any, sophisticated data structures be part of the library?
- Would it be possible to implement sophisticated data structures as part of the library?
- What PL structures can support the making of a better standard library of good data-structures. (yes, logic here is a bit confusing. Think about it this way: if you give the library designer better PL tools, he will be able to design a better datastructures library. Perfection and extensions to the protocol of the standard library would not require any changes to the PL.)
5. Storage

5.2. Arrays

5.2.2. Arrays with integral index types
Efficient but inflexible

Ordinary arrays are formed as mappings from integral types.

**Pros**
- Only values are stored, not indices.
- Simple description of legal indices (defined completely by higher bound, and in some PLs by lower bound as well)
- Efficient access using simple addition:
  - Explicit in C and C++ pointer arithmetic is explicit
    \[ a[i] \equiv *(a+i) \equiv *(i+a) \equiv i[a] \] (2.1)
  - Implicit in, e.g., JAVA, array access it translated to simple machine instructions
  - Range Mapping in, e.g., PASCAL, array access may require subtraction of the first index to compute the actual offset

**Cons**
- When data are sparse, packing techniques are needed.
- Inflexible programming.
Piddles

What are Piddles? (Quotes from the Perl manual)

- Having no good term to describe their object, PDL developers coined the term “piddle” to give a name to their data type.
- A piddle consists of a series of numbers organized as an N-dimensional data set...
- Perl has a general-purpose array object that can hold any type of element...
- Perl arrays allow you to create powerful data structures..., but they are not designed for numerical work. For that, use piddles...
Layout of multidimensional arrays

Two Main Strategies:

- **Multilayered Memory Mapping:**
  1. row-major
  2. column-major

- **Multiple Dereferencing**
5. Storage
5.2. Arrays / 5.2.2. Arrays with integral index types

Row-major layout of 2D arrays (e.g., PASCAL)

Offset of $A_{i,j}$ where $A$ is an $n \times m$ matrix is given by:

$$\text{offset}(A_{i,j}) = (i - 1)m + (j - 1)$$

(2.2)

$A$: a $4 \times 4$ matrix

\[
\begin{array}{cccc}
A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\
A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \\
\end{array}
\]

Column-major layout

\[
\begin{array}{ccccccccccccccc}
A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\end{array}
\]
Column-major layout of 2D arrays (e.g., FORTRAN)

Offset of $A_{i,j}$ where $A$ is an $n \times m$ matrix is given by:

$$\text{offset}(A_{i,j}) = (j - 1)n + (i - 1)$$ (2.3)
“Multiple dereferencing” layout of 2D arrays

Address of \( A_{i,j} \) where \( A \) is a matrix, is given by:

\[
\text{address}(A_{i,j}) = \text{dereferenced}(\text{address}(A) + i - 1) + j
\]  

(2.4)

In C and Java, a 2D array is an array of arrays:
- may be \textbf{null}.
- may be of \textit{any} length
- even length 0 is OK

\[
\begin{array}{cccc}
A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\
A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \\
\end{array}
\]

\[
\begin{array}{cccc}
A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\
A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[0][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
<tr>
<td>a[3]</td>
<td>a[3][0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\
A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \\
\end{array}
\]
Example: triangular array in JAVA

For $A$, an $n \times m$ matrix,

```java
int k = 0;
int[][] iis = new int[][] {
    new int[k++], new int[k++],
    new int[k++], new int[k++],
}; // An array initializer
...
for (int i = 0; i < k; i++)
    for (int j = 0; j < i; j++)
        iis[i][j] = i*j;
```
5. Storage

5.2. Arrays

5.2.3. Type of arrays
Arrays type?

The type of an array of values of type \(\tau\) (first approximation)

- Integer Indexed: \(\text{Integer} \rightarrow \tau\)
- String Indexed: \(\text{String} \rightarrow \tau\)

But, the mapping is only partial; not all possible values of Integer/String indices are mapped into values of type \(\tau\).

Fact (The array type predicament)

To properly define the type of arrays, one needs heavier type theory artillery, which is not really interesting in our course.
Array types in Java

Particularly simple situation

- The type array of $\tau$ includes all arrays of $\tau$, regardless of size.
- All these arrays are assignment compatible.

```java
double[] x, y, z;
x = new double[100];
y = new double[0];
z = x; x = y; y = z;
```
Array types in **Ada**

```ada
type Vector is array (Integer range <> ) of Float;
...

procedure ReadVector(v: out Vector) is ...
  -- Uses v'first and v'last
...

m: Integer := ...
...
a: Vector(1..10);
b: Vector(0..m)
ReadVector(a);
ReadVector(b);
...
a := b;  -- Succeeds only if array b has exactly 10 elements.
```
### 5. Storage

#### 5.3. Variables’ life time

<table>
<thead>
<tr>
<th>5.1</th>
<th>Storage models</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Arrays</td>
</tr>
<tr>
<td>5.3</td>
<td>Variables’ life time</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Simple lifetime</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Storage class</td>
</tr>
<tr>
<td>5.3.3</td>
<td>The heap</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Dangling references</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Heap errors</td>
</tr>
<tr>
<td>5.4</td>
<td>Representation of types in memory</td>
</tr>
<tr>
<td>5.5</td>
<td>Automatic memory management</td>
</tr>
<tr>
<td>5.6</td>
<td>Run time type information</td>
</tr>
</tbody>
</table>
5. Storage

5.3. Variables’ life time

5.3.1. Simple lifetime
Variable lifetime

Definition (Variable lifetime)

The period between allocation of a certain variable and its deallocation is called the lifetime of that variable.

Main varieties: (we will see more in the slides below)

- **Persistent/Permanent**: continues after program terminates
- **Global/Program activation**: while program is running
- **Local/Block activation**: while declaring block is active
- **Heap**: from allocation to explicit deallocation
- **Garbage collected**: from allocation to automatic garbage collection

Lifetime management is important for economic usage of memory.
Persistent variable lifetime

Definition (Persistent variables)

Variables whose lifetime continues after the program terminates are called persistent variables.

Rationale useful for modeling entities such as second storage, files, databases, objects found on web services.

Existence only a few experimental languages offer transparent persistence.

Substitute achieved via I/O operations, e.g., C files: `fopen()`, `fseek()`, `fread()`, `fwrite()`.

Serialization as in JAVA: language/library support the conversion of object into a binary image that can be written on disk or sent over a serial communication line; makes it possible to take objects’ snapshot, save these, and then restore them.
Global vs. local lifetime: simplistic approach

**Global lifetime**  Life of global variables starts at program startup and terminates with the program.
- An *external variable* in C is a variable defined outside of all functions. All external variables have global lifetime.
- In Pascal, all variables defined with the main program are global.

**Local lifetime**  A “local” variable is a variable defined in a function or in a block. Its starts its life when the containing block is activated; its life ends when the block is terminated.

The above terminology is inappropriate since the terms suggests scope as well. However,
- There are “global” variables which are not universally accessible
- There are “local” variables whose lifetime is the same (or almost the same) as the entire program.
More on the simplistic approach
assumes strict block structure and one outermost block

What’s a block?

- **Pascal** functions and procedures
- **ML** ’s let expressions
- **C** and **C++** ’s functions (but also {...} command constructor)

What’s block activation?

- The time interval during which the block is executed
- The same block may be activated more than once
- If $d_1$ and $d_2$ are two durations of activation of two blocks (which may or may not be equal), then, precisely one of the following holds:
  \[ \begin{align*}
  d_1 &= d_2 \\
  d_1 &\subset d_2 \\
  d_2 &\subset d_1 \\
  d_2 \cap d_1 &= \emptyset
  \end{align*} \]
Local & global scopes

Local entity:
- declared in a block
- can be used within the block
- can be used within all nested blocks

Global entity:
- declared in the outermost block
- can be used within all blocks of the program
Variables in the simplistic approach

Global variable:
- Declared in the outer most block
- Lifespan is the same as that of the program

Local variable:
- declared in any other block
- lifespan is the same as block activation
- incarnated each time the block is activated
- may incarnate more than once.
- name may stand in fact for different variables

Location of declaration?
- Usually, to make the compiler’s job easier, declarations are made at the beginning of the block
- However, in C++, JAVA, declarations can be made anywhere in a block
5. Storage

5.3. Variables’ life time

5.3.2. Storage class
Terms “local” & “global” are confusing

Better terms are “automatic” vs. “static variables”

Definition (Storage class in C/C++)

- A storage-class specifier in C or C++, is one of the keywords auto, register, static, extern, typedef, or thread_local (the next few slides will discuss these in greater detail);
- it is used mainly for specifying the lifetime of a variable and its scope.
Better terminology

Can be understood in terms of two of these keywords:

- **Block activation variable** designated by `auto`; allocated on the stack
- **Program activation variable** designated by `static`; allocated in the data segment.
Approximate meaning of C’s storage specifiers

- **auto**: block activation
  
  *block variables with no storage-class specifier default to auto*

- **register**: same as **auto**, but with recommendation to place in a register

- **static**: program activation

- **extern**: program activation
  
  *but declaration must be done somewhere else*

- **typedef**: empty lifetime variables
  
  *exists during compilation, as a template for defining other variables*

- **thread_local**: thread lifetime
  
  *not in the scope of this course*
**static & auto in blocks**

### static in block

```c
/* In the demo version of the software: function undo() can be called only ten times */
void undo() {
    static counter = 10;
    if (--counter == 0)
        return;
    ...
}
```

### auto in block

```c
gcd(int a, auto b) {
    while (a != 0) {
        auto int c = a;
        a = b % a; b = c;
    }
    return b;
}
```

Type of functions with missing type specifier defaults to `int`
### extern & register in blocks

**extern in block**

```c
isPrime(unsigned n) {
    extern isPrimeArray[];
    extern isPrimeArraySize;
    extern isPRIME(unsigned n);
    return
    n < isPrimeArraySize?
    isPrimeArray[n]
    : isPRIME(n);
}
```

**register in block**

```c
isPRIME(register unsigned n) {
    register unsigned d;
    for (d = 2; d*d <= n; d++)
        if (n % d == 0)
            return 0;
    return 1;
}
```

Variables need no type specifier if defined with a storage class; missing type defaults to **int**
Summary: C’s storage specifiers in blocks

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Allowed?</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>missing</td>
<td>✓</td>
<td>same as auto</td>
</tr>
<tr>
<td>auto</td>
<td>✓</td>
<td>block activation (rarely used)</td>
</tr>
<tr>
<td>register</td>
<td>✓</td>
<td>same as auto (but adds a recommendation to the compiler to place in a register)</td>
</tr>
<tr>
<td>static</td>
<td>✓</td>
<td>program activation</td>
</tr>
<tr>
<td>extern</td>
<td>✓</td>
<td>same as static (but must be declared somewhere else)</td>
</tr>
</tbody>
</table>
Examples: C’s storage specifiers at the external level

```c
File a.c

auto x; // 
register double y; // 

/* static storage class: */
static N = 100; /* Accessible only from this file */
static void f(void){} /* Accessible only from this file */

/* extern storage class: */
extern M; /* Defined in some other file */
extern void h(void); /* Defined in some other file */
extern void r(void){} // 

/* missing storage class: */
void g(void) {
    ... } /* Accessible from other files */

int isPrimesArray[] = {
    ... }; /* Accessible from other files */
```
C: access to entities defined in another file

File b.c

```c
extern N = 100;  // X (can you tell (without flipping back) which storage class was used for this, and all other file a.c)
extern void f(void);  // X

/* referred to from file a.c: */
int M = 1000;

/* referred to from file a.c: */
void h(void);

/* Reference to function defined in file a.c: */
extern void g(void)

/* Reference to array defined in file a.c: */
extern isPrimesArray[];
```
Lifetime of static variables in C++ and JAVA

Static variables in C (and PL/I) are used for maintaining state across different activations of a block, regardless of nesting. However, this end is better served with OOP

C++ (tries to maintain C compatibility)
- **block**: from first block activation to the program’s end
- **class**: same as file level
  - **file**: from construction, which occurs sometime before `main()` is called until program end; all such “global” variables are constructed in some order, which is only partially specified by the language’s standard.

JAVA (dynamic loading; truly OO)
- **block**: no `static` variables in JAVA’s functions or blocks.
- **class**: when the class is first used, until program end.
- **file**: no file level variables in JAVA.
C’s storage specifiers at the external (file) level

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Allowed?</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>missing</td>
<td>✓</td>
<td>same as static (but may be referenced via extern from other files)</td>
</tr>
<tr>
<td>auto</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>✓</td>
<td>program activation</td>
</tr>
<tr>
<td>extern</td>
<td>✓</td>
<td>same as static (but must be declared somewhere else without extern or static)</td>
</tr>
</tbody>
</table>
The heap

Definition (Heap variables)

Heap variables are anonymous variables whose life-time spans

From the time they are allocated
- most commonly, directly by the programmer
- at times, as per the runtime environment of the PL (e.g., closures)

Until they are deallocated:
- directly by the programmer, or,
- by the garbage collecting system
Intuition

Think of the *heap* as

- Large, but not infinite, “bank” of memory
- Place from which you can “loan” storage for variables
- If loans are not returned, the bank may become “bankrupt”

The heap can be...

- **Built-in** managed by the PLs runtime systems (as in *Pascal* and *Java*)
- **Library based** Library is
  - Standard (more or less)
  - User replaceable (at least by some “sophisticated” users)
  - examples include *C*, and to some extent, *C++*
Motivation

Program garbage. (* Truly useless program *)
VAR
    p: ^Integer;
Begin
    new(p); (* Allocate a cell *)
    p^ := 5; (* Set its contents *)
    dispose(p); (* Deallocate this cell *)
End.

Why heap variables?

- When the program duration lifetime is inappropriate
- When the contained/disjoint dichotomy of block activation variables is inappropriate
- When memory size is not known in advance
- For realizing data structures such as linked lists, trees, graphs, etc.
Allocation & deallocation

Allocation

**C** Function `malloc()` (library function)

**Pascal** Procedure `new()` (predefined)

**C++'s** Operator `new` (builtin; can be overloaded)

**Java** `new` (keyword)

Deallocation

**C** Function `free()` (library function)

**Pascal** Procedure `dispose()` (predefined)

**C++'s** Operator `delete` (builtin; can be overloaded)

**Java** Automatically, by the GC (GC = Garbage Collector)
Linked list with 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);
odds := cons(1, odds);
```

Diagram:
- `odds`: 1 → 3 → 5 → 7 → |
- `primes`: 2
Access to heap variables

Heap variables are *anonymous*. So, how can they be accessed?

**Definition (Pure reference, referring, and dereferencing)**

- A pure reference or reference for short *(not to be confused now with C++ references)* is a value through which a program may use to indirectly access variable (typically heap variable)
- we say that a reference refers to the variable
- dereferencing is the action of employing a reference to access the variable it refers to

References allow modifications that are more radical than selective updating, and cyclic values which are impossible otherwise.
Many realizations of references

**Address** most commonly, references are nothing but memory addresses, in which case, they are called *pointers*.

**Offset** references may be implemented as offsets from a fixed address.

**Array index** in a language that forbids manipulation of memory addresses, references may be realized as array indices.

**Handle** Index into an array which contains the actual pointer.

**Smart pointer** an abstract data type that extends the notion of pointers, while providing services such as
  - computing frequency of use
  - reference counting
  - lazy copying
  - caching
  - legality of access checking
  - …
## The C++ “references vs. pointers” confusion

<table>
<thead>
<tr>
<th>C++</th>
<th>Java (and many other PLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer, i.e., pointer variable</td>
<td>Pointer?</td>
</tr>
<tr>
<td></td>
<td>no such beast;</td>
</tr>
<tr>
<td></td>
<td>sometimes used as a synonym for “reference”</td>
</tr>
<tr>
<td></td>
<td>Must be explicitly “dereferenced”</td>
</tr>
<tr>
<td></td>
<td>May not be “0”</td>
</tr>
<tr>
<td></td>
<td>Must point to a variable</td>
</tr>
<tr>
<td></td>
<td>Cannot be changed</td>
</tr>
<tr>
<td></td>
<td>No “dereferencing” prior to use.</td>
</tr>
<tr>
<td></td>
<td>Is pure reference</td>
</tr>
<tr>
<td>Reference, i.e., reference variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>May be “null”, or point to a variable</td>
</tr>
<tr>
<td></td>
<td>Can change</td>
</tr>
<tr>
<td></td>
<td>No “dereferencing” prior to use.</td>
</tr>
<tr>
<td></td>
<td>Is not pure reference</td>
</tr>
</tbody>
</table>
The “null” pointer

- Strictly speaking, the “pure” definition requires that all references provide access to some variable.
- Still, it is useful to have references which “refer to nothing”; e.g., for designating the end of a linked list.
- It is possible to realize “refer to nothing” as reference to a special variable.
- It is more convenient to allow a special, illegal value of references instead. This value is known in different languages as null, nil, void, nullptr (A new C++ keyword), 0, etc.
- In Java and many other languages, references are disjoint sum of “pure references” and Unit.
- C++’s references are nothing but immutable, pure references.
5. **Storage**

5.3. Variables’ life time

5.3.4. **Dangling references**
Dangling references

Definition (Dangling reference)

A dangling reference is a reference to a variable whose lifetime has ended, e.g., a variable which has been deallocated

Lifetime may end as a result of...

- Termination of containing block
- Deallocation
How are dangling references created?

I. Freed memory

*a deallocated heap variable:

```c
char *p = malloc(100);
strcpy(p, "Hello, World!\n");
free(p);
// p is dangling
strcpy(p, p + 5); // X
```

II. Reference to stack

*reference to a “dead” automatic variable:

```c
char *f() {
    char a[100];
    return &a;
}
char *s = f();
// s is dangling
strcpy(s, "Hello, World!\n"); // X
```

III. Inner functions

*Activating a function outside the enclosing block in which it was defined, in the case that the function uses variables local to the block

```c
// Provide name for function type:
typedef void (*F)(void);
// Forward declaration
F f();
// of function returning a function
F h = f();
// May access a dangling reference
F f() {
    char a[100];
    // only Gnu-C allows inner functions
    void g(void) {
        // a is dangling if g
        // is called from outside f
        strcpy(a, "Hello, World!\n"); // X
    }
    return g;
}
```
## Language protection against dangling references

<table>
<thead>
<tr>
<th></th>
<th>Freed memory</th>
<th>Stack reference</th>
<th>Inner functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>programmer's responsibility</td>
<td></td>
<td>programmer's responsibility</td>
<td>no inner functions</td>
</tr>
<tr>
<td><strong>Gnu-C</strong></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>programmer's responsibility</td>
<td></td>
<td>programmer's responsibility</td>
<td>programmer's responsibility</td>
</tr>
<tr>
<td><strong>Pascal</strong></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>programmer's responsibility</td>
<td>cannot take the address of stack variables</td>
<td>inner functions cannot leak</td>
<td></td>
</tr>
<tr>
<td><strong>Java</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>programmer never deallocates memory, thanks to garbage collection</td>
<td>objects: always on the heap; stack: scalars or references to objects; no references to stack</td>
<td>functions are not first class; their address cannot be taken</td>
<td></td>
</tr>
</tbody>
</table>
Quiz: What’s dangling here?

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

struct N {
    struct N *n;
    char *q;
    int a;
} *q;

int foo(void) {
    N *p = malloc(sizeof *p);
    p->a = 42;
    p->q = "life, universe, everything";
    p->n = (struct N *)p;
    q = p;
    free(p);
    return 1;
}

int main() {
    free(strcpy((char *)malloc(20), foo() + "Hello, World\n"));
    (void) printf("q=%p\n", q);
    (void) printf("q->a=%d\n", q->a);
    (void) printf("q->q=%s\n", q->q);
    return 0;
}
```
What’s the penalty of accessing a dangling reference

Well, it depends whether you are lucky or not…

- **Lucky** Immediate program crash
- **Unlucky** Program crashes, but not immediately
- **Extremely unlucky** Program does not crash while testing; it just has a bug which stays dormant until field trial!

*In our case, the output is*

**Output of our dangling reference program**

```
q=0x2314010
q->a=42
Segmentation fault (core dumped)
```

We were quite lucky, however, if we would not have accessed the `q` field, the bug would not have been detected!
Stack corruption via dangling reference into an automatic variable

```c
#include <string.h>
#include <stdio.h>

char *f(void); // Forward declaration

main() {
    char *hell = f();
    printf("%s\n", hell);
    return 0;
}

cchar *f(void) {
    char s[1<<9];
    strcpy(s, "Hello, World\n");
    return s;
}
```

The output looks almost right, but the stack is clearly corrupted.
5. **Storage**

5.3. **Variables’ life time**

5.3.5. *Heap errors*

<table>
<thead>
<tr>
<th>5.1</th>
<th>Storage models</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Arrays</td>
</tr>
<tr>
<td><strong>5.3</strong></td>
<td><strong>Variables’ life time</strong></td>
</tr>
<tr>
<td>5.3.1</td>
<td>Simple lifetime</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Storage class</td>
</tr>
<tr>
<td>5.3.3</td>
<td>The heap</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Dangling references</td>
</tr>
<tr>
<td><strong>5.3.5</strong></td>
<td><strong>Heap errors</strong></td>
</tr>
<tr>
<td>5.4</td>
<td>Representation of types in memory</td>
</tr>
<tr>
<td>5.5</td>
<td>Automatic memory management</td>
</tr>
<tr>
<td>5.6</td>
<td>Run time type information</td>
</tr>
</tbody>
</table>
Dangling references

Recall: “a dangling reference is a reference to a variable whose lifetime has ended... Lifetime may end as a result of...

- Termination of containing block
- Deallocation
Memory leak

**Definition (Memory leak)**

A memory leak occurs when a variable is not deallocated prior to the termination of the lifetime of all of its references.

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct N {
    int a[1000000000];
} N;

N *t(N *n){
    N *p = malloc(sizeof *p);
    if (p != 0) return p;
    perror("OOPS");
    exit(1);
}

int main() {
    int i;
    N *s = 0;
    for (i = 0 ; i < 1 << 30; i++)
        printf("%5d) %p
",i,s=t(s));
    return 0;
}
```

Can you pinpoint the leak?
Output of the above program

0) 0x7f7b8adf0010
  1) 0x7f7a9c73d010
  2) 0x7f79ae08a010
  ...
1000) 0x7bd8382b8010
1001) 0x7bd749c05010
1002) 0x7bd65b552010
  ...
10000) 0x5b1a4fdc0010
10001) 0x5b196170d010
10002) 0x5b187305a010
  ...
100000) 0x7ffc1105b010
100001) 0x7ffcff70e010
100002) 0x7ffdeddc1010
...
35180) 0x7ffc1105b010
35181) 0x7ffcf70e010
35182) 0x7ffdeddc1010

OOPS: Cannot allocate memory
Simple heap management with linked list of free blocks

(we maintain the invariant that free regions are in ascending addresses)

Allocation request Traverse the list, searching for a region large enough to accommodate the request, using a “first fit”, “best fit” or “worst fit” strategy.

Exact match the region is returned to the client and removed from the list

Non-exact match the region is split into two:

1 a sub-region is of the appropriate size is returned to the client;
2 the other sub-region is kept in the list.

Deallocation request Add the region to the list; if there are no gaps, with the previous/next region, the two are merged.

Most real life implementations use much more sophisticated data structures.
<< < HEAD ====== >> >
5. **Storage**

5.4. Representation of types in memory

5.1 Storage models
5.2 Arrays
5.3 Variables' life time
5.4 Representation of types in memory
5.4.1 Simple representation of types in memory
5.4.2 Reference vs. value semantics
5.4.3 Shared representation & lazy copy
5.4.4 Value vs. reference semantics in various PLs
5.5 Automatic memory management
5.6 Run time type information
5. Storage

5.4. Representation of types in memory

5.4.1. Simple representation of types in memory

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage

5.1 Storage models
5.2 Arrays
5.3 Variables’ life time
5.4 Representation of types in memory
5.4.1 Simple representation of types in memory
5.4.2 Reference vs. value semantics
5.4.3 Shared representation & lazy copy
5.4.4 Value vs. reference semantics in various PLs
5.5 Automatic memory management
5.6 Run time type information
Purpose of type?

We have values; **why do we need types?**

- **Taxonomy** of values; describe data effectively
- **Legality** determine set of legal operations on values (prevent nonsensical operations, e.g., multiply a pointer by a set)
- **Semantics** determine semantics of operations on values

But, also, defines program-machine interface:

**Program** $\Rightarrow$ **Machine** how to represent values on different machines

**Machine** $\Rightarrow$ **Program** how to represent values on different machines
PLs policy for type representation

The selection of atomic types also determines a policy of mapping these types into memory.

Policy for representation of types and values

A PL $\mathcal{L}$ sets a policy $P_{\mathcal{L}}$ for the representation of $\forall \mathcal{L}$ on machines $M_1, M_2, \ldots$,

$$P_{\mathcal{L}} : \forall \mathcal{L} \rightarrow \{M_1, M_2, \ldots\} :$$ (4.1)

most often, employs the recursive nature of the type system

- How to represent atomic types?
- How to represent type constructors?
Implicit contract of type representation policy

Policy $P_L$

$$P_L : \forall L \rightarrow \{M_1, M_2, \ldots, \}$$

sets a *contract* between two parties:

**Implementer of $L$** Given a machine $M_i$ select mapping $\forall L \rightarrow M_i$ which is
- Most efficient
- Compliant with $P_L$

**Programmer using $L$** Write a program which is
- Most efficient
- Does not depend on the specifics of $\forall L \rightarrow M_i$
Exception 1: non-primitive atomic types

The main non-primitive atomic type is *enumeration*

- Usually mapped to integer values
- PLs often stay silent regarding representation
- **Java**, *enums* are ordinal values with their own unique operations
- **Java** even allows different operations for different enumerands in the same enumeration
Exception II: pointer values

**Representation of pointer values**

*Most PL map pointers to hardware address types*

**Issues:**

- Sometime the hardware supports several address types, e.g., on X86 architectures, with **near** and **far** pointers.
- Many modern PLs (include **JAVA** and **ML**) have “smart” pointers, which cannot be mapped directly to hardware.
Exception III: weird atomicity

In most cases:

- values of atomic types are atomic values
- atomic values belong to atomic types

**Anomaly I**

Compound types whose values are atomic.

**Pointers**

**Anomaly II**

Atomic types whose values are compound.

- Values of *atomic type string* where it exists, are non-atomic

Hardware mapping policy needs to deal with these two.
5. Storage

5.4. Representation of types in memory

5.4.2. Reference vs. value semantics
### Variables: reference vs. value semantics

#### Value Semantics.
- **Variable** contains the actual value.
- **C, C++**
- **JAVA** for builtin, atomic types

#### Reference Semantics.
- **Variable** contains a reference to a value which is stored elsewhere.
- **C, C++, if pointers or references are used**
- **JAVA** for all other types, including arrays
- **Most modern languages**
Values vs. reference semantics in JAVA

The basic type system of JAVA is defined by:

- 8 atomic types: byte, short, int, long, float, double, boolean, char
- 1 pseudo type: void
- 4 type constructors:
  - array
  - class
  - interface
  - enum

- Precisely 8 types in JAVA follow value semantics
- All the rest are reference semantics
Wrapper classes

- When generics were introduced to Java, it was discovered that the implementation was much simpler for reference types.
- The 8 value types did not justify extra machinery:
- Instead, the Java library introduced reference type equivalents

  Integral types  Byte, Short, Integer, Long
  Floating point types  Float, Double
  Other types  Boolean, Character
  Unit types  Void

```
List<double> ds1;  // X compilation error;
// type double is not
// a reference type

List<Double> ds2;  // ✓ works fine;
// type Double is
// a reference type
```
“Integer” vs. “int” in JAVA

- Each wrapper class (except for Void) wraps a value of the corresponding primitive type.
- Wrapper types are almost fully interchangeable with their primitive equivalents:

```java
int v = 3; // Primitive type
Integer r = new Integer(a); // Wrapper type
v = r.intValue(); // Explicit conversion
v = r; // auto un-boxing
r = v; // auto boxing
```
Boxing

Auto boxing  Coercion from, e.g., \texttt{int} to \texttt{Integer}

Auto unboxing  Coercion from, e.g., \texttt{Integer} to \texttt{int}

- Type \texttt{Integer} also includes value \texttt{null}
- Type \texttt{int} does not include value \texttt{null}
- The following will generate \texttt{RuntimeException}

```java
Double dd = null;
double d = dd;
```
The OO terminology

Objects?

- **OO languages often use the term “object”**...
- **the term propagates also to non-OO PLs**
- **means (usually) a “variable” whose contents has an i.d.**
- **total inspection of this object, yields a value with an i.d.,**

Object’s **i.d.**,

- cannot be changed by user
- two object with the same contents, still have a distinct **i.d.**
C: value semantic of assignment

A “date” record in C

typedef struct Date {
    int year, month, day;
} Date;

Initializing two variables of this type:
Date today = \{2017,12,13\};
Date tomorrow = \{2017,12,14\};

today = tomorrow;

Before assignment

<table>
<thead>
<tr>
<th>today</th>
<th>tomorrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017 12 13</td>
<td>2017 12 14</td>
</tr>
<tr>
<td>year month day</td>
<td>year month day</td>
</tr>
</tbody>
</table>

After assignment

<table>
<thead>
<tr>
<th>today</th>
<th>tomorrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017 12 14</td>
<td>2017 12 14</td>
</tr>
<tr>
<td>year month day</td>
<td>year month day</td>
</tr>
</tbody>
</table>
Reference semantic of assignment in JAVA

A “date” record in JAVA

class Date {
    Date(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }
    int year, month, day;
}

Date today =
    new Date(2017, 12, 13);

Date tomorrow =
    new Date(2017, 12, 14);

today = tomorrow;

Before assignment

today

2017 12 13
year month day

tomorrow

2017 12 14
year month day

After assignment

today

2017 12 12
year month day

tomorrow

2017 12 14
year month day
C++ vs. Java (Can you detect and explain all the syntactical differences between the two languages?)

**C++**: Value semantics

```cpp
class Date {
public:
    int year, month, day;
    Date(int year, int month, int day) {
        this->year = year;
        this->month = month;
        this->day = day;
    }

    Date today(2017, 12, 13);
    Date tomorrow(2017, 12, 14);

    today = tomorrow;
    tomorrow.year = 3025;
    cout << today.year;
}
```

**Java**: Reference semantics

```java
class Date {
    int year, month, day;
    Date(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    Date today = new Date(2017, 12, 13);
    Date tomorrow = new Date(2017, 12, 14);

    today = tomorrow;
    tomorrow.year = 3025;
    System.out.print(today.year);
```

**Summary**
- **C++**: Uses `this->` notation: `value` semantics
- **Java**: Uses `.` notation: `reference` semantics

- **Uses Operator Overloading**: No operator overloading in Java.
- **Variable Contains Value**: C++
- **Variable Refers to Value**: Java
Comparing the two semantics

**C++: Value semantic**

```cpp
// Creating values for today and tomorrow
Date today = Date(2017, 12, 13);
Date tomorrow = Date(2017, 12, 14);
// Assigning variable tomorrow to today
today = tomorrow;
tomorrow.year = 3025;
cout << today.year;
```

**Java: Reference Semantic**

```java
// Creating values for today and tomorrow
Date today = new Date(2017, 12, 13);
Date tomorrow = new Date(2017, 12, 14);
// Assigning variable tomorrow to today
today = tomorrow;
tomorrow.year = 3025;
System.out.print(today.year);
```

**Output is...**

**C++**

```
2017 12 14
```

**Java**

```
3025 12 14
```

**Output is...**

C++: 2017

Java: 3025
5. Storage

5.4. Representation of types in memory

5.4.3. Shared representation & lazy copy
Which semantic does ML use?

**Answer**

*The programmer shouldn’t care and cannot know!*

- *It looks like value semantics.* In reality, ML, Lisp and many other languages use *value semantics*, in the sense that the programmer cannot observe any “references” in the program.

- *Implementation is with references.* Behind the scenes, memory and time is saved by using references.
Efficient *LISP* implementation with references

- Let $\alpha, \beta$ be two large $S$-expressions.
- `(setq a \alpha)`
- `(setq b \beta)`
- `(cons a b)`
- `(setq c (cons a b))`

```
<table>
<thead>
<tr>
<th>α</th>
<th>Some large $S$-expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Some other large $S$-expression</td>
</tr>
</tbody>
</table>
```

\(\alpha\)

\(\beta\)
Generalization

Definition (Lazy copying)

Generalizing the \texttt{Lisp} approach, lazy copying is an implementation technique of value semantics, where a copy of a large object is made by creating a new reference to it. The actual copy operation is made when (and if) the source or the destination variables are modified.

- Generalizes the \texttt{Lisp} approach
- Support for languages which permits mutation of values.
- Many extensions of pure-\texttt{Lisp} allow such mutation
- Mutation is the bread and butter of imperative programming.
- Conceptually similar to “copy on write” in memory management
Reference semantic in C++?

Of course!
You just have to be explicit about it!

```cpp
class Date {...};

// Storing references to newly allocated values
// of today and tomorrow
Date *today = new Date(2017, 12, 13);
Date *tomorrow = new Date(2017, 12, 14);
today = tomorrow; // Leak!
tomorrow->year = 3025;
cout << today->year;
delete tomorrow;
delete today; // Heap corruption?
```

Output is... 3025
Value semantic in Java?

Of course!
You just have to be explicit about it!

class Date implements Cloneable {
    :
}

today = (Date) tomorrow.clone();
tomorrow.year = 3025;
System.out.println(today.year);

Date today = new Date(2017, 12, 13);
Date tomorrow = new Date(2017, 12, 14);

Output is... 2017
Assignment in reference semantic languages?

But, ... what does

(Date) tomorrow.clone();

actually mean?

More generally, in any reference semantic programming language:

- Given two variables, \( a \) and \( b \),
- each containing a reference to a value,
- which may include references to a network of variables,
- and an assignment command

\[ a := b; \]

- what's going to happen?
Assignment in reference semantic languages?

**Reference assignment**

*Only the reference is copied*

![Diagram illustrating reference assignment]

**Deep clone**

*The whole network of variables accessible from b is duplicated, and assigned to a*

![Diagram illustrating deep clone]

**Shallow copy**

*Only the referenced value is copied*

![Diagram illustrating shallow copy]
Shallow clone: yet another semantics

The variable itself is cloned, but all the references inside it are copied, rather than being cloned.

Before assignment

After assignment
Assignment strategies side by side

Ref. assignment

Shallow copy

Shallow clone

Deep clone
Properties of assignment strategies

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Null pointer assignment?</th>
<th>Memory allocation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Never</td>
<td>✗</td>
</tr>
<tr>
<td>assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shallow copy</td>
<td>Maybe</td>
<td>✗</td>
</tr>
<tr>
<td>Shallow clone</td>
<td>Never</td>
<td>✓ (bounded)</td>
</tr>
<tr>
<td>Deep clone</td>
<td>Never</td>
<td>✓ (unbounded)</td>
</tr>
</tbody>
</table>
5. Storage

5.4. Representation of types in memory

5.4.4. Value vs. reference semantics in various PLs
What does **Java** `clone()` do?

**Runtime Exception**
- if the class *does not implement* interface `Cloneable`

**Shallow clone**
- if the class implements interface `Cloneable`, and
- the programmer *does not override* the default `clone()` method.

**Whatever**
- if the class implements interface `Cloneable`, and
- the programmer *overrides* the default `clone` method
  - *in whatever way he likes.*

**Deep clone**
- if the class implements interface `Cloneable`, and
- the programmer overrides the default `clone` method, and
  - *correctly implements a* "deep clone" *semantic*
Working knowledge of semantics?

Typical exam question: Read the documentation of a particular language feature, and determine which semantic it uses.

Feature could be “assignment” (of a particular kind of variable), “library function”, and even “equality testing”: comparison of components one by one vs. meager comparison of the references.
Overloading the assignment operator in C++

```cpp
class Date {...};
Date today(2017, 12, 13);
Date tomorrow(2017, 12, 14);
today = tomorrow; // Call the assignment operator
```

And, what does the assignment operator do? (If you think you are smart, please repeat for copy constructor)

User Defined

Whatever...

- Can anyone really understand programmers’ mind?
- The author of these slides (at least) cannot.

Default Behavior

Not so clear...

- recursively apply assignment operator on each of the fields
- Non-user defined types: shallow copy.
- The default assignment operator will typically be shallow copy.
Case study: assignment & copy in **EIFFEL**

**Value Semantic**  Atomic types, such as **Char**, **Integer**, **Real**, and **Boolean**, and object attributes marked as **expanded**

**Reference Semantic**  Everything else

--- Reference (identity) operations:
- `a := b`  -- Reference assignment
- `a = b`  -- Reference equality testing
- `a /= b`  -- Reference inequality testing

--- Shallow (state) operations:
- `a.copy(b)`  -- Attribute by attribute
  -- shallow copy
- `a := clone(b)`  -- Create new cloned object
- `equal(a,b)`  -- Attribute by attribute
  -- comparison

--- Deep (state) operations:
- `a.deep_copy(b)`  -- Attribute by attribute
  --- copy
  --- and cloning of inner objects
- `a := deep_clone(b)`  -- Create a full clone of
  --- a complex structure
- `deep_equal(a,b)`  -- Attribute by attribute
  --- recursive comparison
More general working knowledge (Typical exam question, if you like it phrased this way…)

Language Design Question

Suppose that Pascal had a list type constructor

```pseudo_pascal
VAR
primes, odds: list of Integer;
```

What does `primes := odds` mean?

**Reference Copying**
- Inconsistent with arrays, records and primitive types
- Pointers in disguise
- Selective updates to one will affect the other

**Value Copying**
- Natural, but inefficient
- Possible solutions: prohibit selective update (as in Lisp), or lazy copying
Semantics in some contemporary PLs

**Value Semantic**  
Pascal, Lisp, ML, Prolog

**Reference Semantic**  
Java, Smalltalk

**Mixed Semantic**  
Eiffel, C, C++

Most languages have some kind of a mix:

- In Java, primitive types have value semantic
- There are hacks in Lisp that allow reference semantic
- References in ML allow reference semantic
- Eiffel has expanded types
- C# has “non-nullable” types.

In most cases, a conclusive judgment “value/reference semantic” for an entire language is plain wrong. (as are some of the sweeping judgments made in this slide...)
Overview: semantics of assignment

- Assignment semantics is defined by the language design:
  - C structures follow value semantics.
  - C used to place restrictions on passing structures by value.
  - Arrays cannot be assigned.
  - Pointers are used to implement reference semantics.
  - Java follows value semantics for primitive types.

- Value semantics may be slower
- Reference semantics may lead to sharing problems.
- Reference semantics is more expressive.
5. Storage

5.5. Automatic memory management
Memory management

- Stack Based
- Escape Analysis
- Manual
- Dangling Reference
- Automatic
- Memory Leak
- Garbage Collection
- Memory Compaction
- Handles
- Reference Counting
- Copying Collector
- Algorithms
- Mark and Sweep
- Cautions
- Null Pointer Reference
- Memory Leak
Reference counting

**Idea** a “reference count” (RC) field in every variable

**Invariant**
- RC is the number of references to the variable.
- The RC of all live variable is positive

**Initially** In allocation such as (No other allocation command makes sense):

```java
Thingy t = new Thingy(); // JAVA syntax
```

Set RC of the newly created **Thingy** to 1.

**Maintenance**

```java
Object o1 = new Object();
// Denote the newly allocated object by O1;
// Set RC(O1) ← 1;
Object o2 = new Object();
// Denote the newly allocated object by O2;
// Set RC(O2) ← 1
...
Object o2 = o1 // RC(O1)++; RC(O2) --;
```

**De-allocation** After each *decrement*, if \( \text{RC}(O) = 0 \): (i) de-allocate \( O \); (ii) decrement RC for all children of \( O \); and, (iii) recursively de-allocate objects whose \( \text{RC}=0 \).
Pros & cons of reference counting

Pros
- predictable performance
- smooth execution without interruptions
- Implementable in Manual Memory Management System via smart pointers, or even as part of the language semantics. Automatic Memory Management System as part of the garbage collection system.
- cost is proportional to actual computation, not to memory size

Cons
- Cannot deal with circular structures
- Is generally slow, incurring a huge “write barrier” (the amount of work that needs to be done in each memory write)

Fact (Write barrier)

The formidable write barrier excludes the universal application of RC for memory management
What is garbage collection?

**Definition (Mark & sweep GC algorithm)**

*Invented by John McCarthy around 1959 as an enabling technology for Lisp implementation, Garbage Collection (GC) is a part of the program semantics and runtime, which automatically claims back all unused memory.*

In simple words, de-allocation becomes the responsibility of the PL’s runtime system, rather than the programmer’s.

- **Programmer never de-allocates memory**

- When memory becomes scarce, a GC procedure is applied to collect all unused variables

- *Mark & sweep*: the simplest GC algorithm found in JAVA, SMALLTALK, PYTHON, LISP, ML, HASKELL, and most functional, or modern OO languages.
Why garbage collection?

GC prevents

- Dangling references
- Memory leak
- Heap corruption
- Heap de-fragmentation (with a compacting collector).

Also, GC makes first-class functions value possible.
This is our storage bank, which contains many “cells”. Our interest, though, lies only with “allocated” cells. An allocated cell is called a variable. Some variables follow value semantics; others contain references. Some belong in the runtime stack, others are "global"; the rest are heap allocated. The heap is primarily a list of free blocks. But, the heap also maintains another list, which keeps references to all heap allocated cells! So, we have a list of free blocks and a list of allocated cells. Together, the two lists make the Heap Data Structure.

Now, our variables reference each other, and have many null references, which we will not always show. For garbage collection, we define the Root Set, which contains global variables, and the runtime stack. Starting from the root set, we conduct a mark phase. We first mark all variables in the root set, and then follow their references, to mark variables referenced from the root set. Again, we follow references, to mark variables two references away from the root. We keep following references, until we mark all variables reachable from the root set. Consider now on the entire set of variables. Which variables are garbage? In the sweep phase, we collect all variable blocks which are not marked. Recall the "List of Allocated Cells"? Let's iterate over it! Start at the first node, and iterate, and iterate, and iterate, 1st cell for recycling, and iterate, 2nd cell for recycling, and iterate, and iterate, and iterate, and iterate, 3rd cell for recycling, and iterate, 4th cell for recycling, until we are done! All that remains is, to remove the cells destined for recycling from the "List of Allocated Cells", to claim back the memory they occupy, and to add these memory blocks to the back to the "List of Free Cells". This diagram depicts the main points.
Summary: mark & sweep garbage collection

Mark  mark all cells as unused

Sweep  unmark all cells in use (stack, global variables), and cells which can be accessed, directly or indirectly, from these

Release  all cells which remain marked
Delicate issues of the marking process

- Do not visit an object more than once
- Do not get stuck in a loop.
- Typical implementations:
  - Breadth-first search
  - Depth-first search
- Marking:
  - Can be done by “raising” a bit in each object
  - More efficient procedure:
    - Initially, all objects are “0”
    - In first collection, marking is by changing the bit to “1”
    - In second collection, marking is by changing the bit to “0”
    - In third collection, marking is by changing the bit to “1”
Stop & copy garbage collection

- Divide the heap into two regions:
  - Region I takes all allocations
  - Region II is put on hold
- When region I is exhausted, copy live (reachable) variables to region II
- Switch the roles of the two regions
Defragmentation

- Can be done whenever the GC detects memory fragmentation:
  - Lots of memory is available
  - All memory is fragmented into small consecutive chunks.
  - Program requires a large consecutive memory, e.g., for array.

- Can be done in each collection cycle:
  - Presumably slower
  - Often performs better due to caching and “locality of reference”.
Predicaments of garbage collection

**Memory/Time Resources** could be saved using programmers’ knowledge.

**Decreased Performance** of the [Real] core program

**Uneven Performance** with “embarrassing pauses” for GC cycles

**Unpredictable Performance** the program can never know when a GC cycle may start

**Not for Real Time** which requires predictable performance

**Not for Transactions** a transaction may time out with no good reason

**Hinder Interactiveness** pauses can lead to user abandonment

**Incompatible with “Resource Allocation Is Initialization”** cannot rely on the destructor of a file object to close the file
Some responses to these predicaments

**Generational GC** collects variables at the nursery first, where mortality is high

**Incremental GC** Can perform some computation and resume it later.

**Concurrent GC** Can run concurrently to the program.

**Realtime GC** Obeys time constraints

Concurrency, predictability, etc., always incur a performance toll.
Memory leak in garbage collection?

- GC can only claim reachable variables
- If a programmer forgets to nullify references, then a pseudo memory leak may occur

Define a class `Leak`

```java
public class Leak {
    ...
}
```

whose contents is:

```java
private Leak next;
private int[] data;

private Leak(Leak next) {
    this.next = next;
    this.data = new int[1<<25];
}

private static Leak cons(Leak l) {
    return new Leak(l);
}
```

```java
public static void main(String[] args) {
    Leak l = new Leak(null);
    final Runtime r = Runtime.getRuntime();
    for (int i = 0; i < 100; ++i) {
        System.out.println(i + " : 
            r.freeMemory());
        l = cons(l);
    }
}
```
Output of the above program

0: 123021432
1: 123086952
2: 123152472
3: 123217992
4: 123283512
5: 123349032
6: 123414552
7: 123480072
8: 123545592
9: 70526952

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at Leak.<init>(Leak.java:16)
  at Leak.cons(Leak.java:12)
  at Leak.main(Leak.java:8)
Semantical memory leak (Note that the previous example exhausted memory for the sake of demonstration; it did not really create semantic garbage. )

Definition (Semantical garbage)

A variable which the program will never use again, but still keeps a reference to it, is called semantic garbage.

class Huge {
    Huge() {
        // Constructor:
        // Allocates lots of data and stores it in the newly created object
    }
}

void f() {
    Huge semanticGarbage = new Huge();
    heavy.computation(new Indeed(100));
    System.exit(1);
}

The semantic garbage predicament

All sophisticated GC algorithms contend in vain against semantic garbage
GC & the stack: escape analysis

- GC is always slower than stack-based memory management.
- In a pure GC, there are no automatic variables.
- In Java, variables can be:
  - **Stack allocated** primitive types: `int`, `double`, `boolean` etc.
    - No references to primitive types
  - Stack allocated References to classes and arrays.
    - No references to primitive types
  - **Heap allocated** Classes and arrays (accessed only by references)

Seemingly Innocent Program

```java
void foo() {
    int a[] = new int[1 << 20];
    List<Integer> b = new ArrayList<Integer>();
    // does a gets assigned to global variables?
}
```

With escape analysis a smart compiler can determine that variables a and b never “escape” function `foo()`, and then can be safely claimed when this function terminates.
5. Storage

5.6. Run time type information
The challenge of deep clone

“Algorithm” for Deep Clone:

- Start from current value.
- Traverse the network of values accessible from it.
- Duplicate this network.

How should we “traverse” the network?

**Definition (Network Traversal: breadth- (or depth-) first search)**

_In Each Value we Visit:_

- Mark the value as “visited”
- Proceed to all values it references
But, there is a catch…

Definition (Network Traversal: breadth- (or depth-) first search)

In Each Value we Visit:
- Mark the value as “visited”
- Proceed to all values it references

The challenge: When we reach a value, we do not know what’s in it!
Memory $=$ bits & bytes!

To understand the difficulty better, we need to take a second look at:

- Bits
- Bytes
- Values
- Types
- Memory representation of values
- The interpretation of memory representation
Example: different Interpretations of a single byte

```
#include <stdlib.h>
#include <stdio.h>

main() {
    const void *p = malloc(1);
    *(unsigned char *)p = 0b1001011;
    printf("As integer: '%d'; 
    
    printf("as character: '%c'

As integer: '75'; as character: 'K'
```
Example: different interpretations of a 16 bits word

```
#include <stdlib.h>
#include <stdio.h>

main() {
    const void *p = malloc(2);
    *(unsigned short *)p = 0b1101011100001001;
    printf("As signed integer: '%d'\n", *(signed short *)p);
    printf("As unsigned integer: '%d\n", *(unsigned short *)p);
    printf("As an array: '(%d,%d)\n", 0[(char *)p], 1[(char *)p]);
}
```

As signed integer: '-10487'
As unsigned integer: '55049'
As an array: '(9,-41)'

Example: different interpretations of 32 bits word

```c
// A more civilized way to name integer values:
enum {
    // How many bits for index into pool:
    LG2_POOLSIZE = 14,
    // How many bits for storing car/cdr kind:
    KIND_SIZE = 2
};
enum kind { NIL, ATOM, STRING, INTEGER};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
};
```

<table>
<thead>
<tr>
<th>tag</th>
<th>variant part</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2b</td>
</tr>
<tr>
<td></td>
<td>14b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>variant part</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2b</td>
</tr>
<tr>
<td></td>
<td>14b</td>
</tr>
</tbody>
</table>

CONS: machine word (32b)
The layout of a C structure

The same memory block could be interpreted in many different ways. Here is a 16 bytes block, which can be interpreted as struct T0, as struct T1, or, as struct T2.

```
struct T0 {
    float x;
    struct T1 *p;
    char s[4];
    struct T2 *q;
};
```

```
struct T1 {
    char s[4];
    float x;
    struct T2 *q;
    struct T0 *p;
};
```

```
struct T2 {
    struct T0 *p;
    char s[4];
    struct T1 *q;
    float x;
};
```
Summary: the “meaning” of bits and bytes

A value is represented in memory as a sequence of bits and bytes.

Components:
- Integers
- Floating point values
- Characters
- References
- Arrays
- Sets in bit mask representation.
- etc.

Deciphering a Value

- The values’ type is the key
- It gives meaning to the bit representation.

Information provided by type:
- Value’s length
- Partitioning into sections
- Appropriate way of interpreting each section
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```

```
struct T2 { struct T0 *p; char s[4]; struct T1 *q; float x; };
```
Interpreting bit and bytes as values of a type

**Definition (Static typing)**

The compiler knows the “deciphering key”, and it generates code based on this information.

**Definition (Dynamic typing)**

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

RTTI field may contain all the type information

more commonly, RTTI is a reference to a “type descriptor” shared by all values of the type.
Designing an algorithm for traversing values

Can we use static type information?

*No!!!*

- The network of objects typically contains values of very many distinct types
- The traversal algorithm should know
  - the type of each visited value,
  - the types of each of the values it references
- It is impractical to generate a different traversal algorithm for each input program as per the different that occur in it.
RTTI is the answer!

Definition (Run-time type information)

Run-time type information (or RTTI for short) is a tag attached to each value, which specifies its type.

Application of RTTI in different kinds of PLs:

Statically Typed
- Deep cloning,
- Garbage collection, and,
- Serialization.

Dynamically Typed
- Deep Cloning,
- Garbage Collection,
- Serialization, and,
- Run time type checks
C, C++, & RTTI

- As a result of the “no hidden cost” language principle, C does not and cannot have RTTI.
- As a result, C cannot have general purpose GC, serialization, cloning or any deep operations.
- Due to the “C-compatibility at almost all costs” language principle, C++ does not and cannot have RTTI.
- As a result, C++ cannot have general purpose GC, serialization, cloning or any deep operations.
- C++ has a limited form of RTTI for the implementation of virtual functions.
- More on these mysterious “vptr” and “vtbl” in our OOP course.
## Use of RTTI in the implementation of different PLs

- Consider a variable `today` which references an object with (say) three fields: `year`, `month`, `year`.
- How is `today.day=35` being implemented?

<table>
<thead>
<tr>
<th>prog. lang.</th>
<th>C</th>
<th>Java</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>syntax</strong></td>
<td>today-&gt;day=35</td>
<td>today.day=35</td>
<td>today.day=35</td>
</tr>
<tr>
<td>static typing</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>dynamic typing</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RTTI</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>type punning</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Implementation**

- 1. dereference `today`
- 2. advance by `off(day)`
- 3. update field

- 1. dereference `today`
- 2. ignore RTTI
- 3. advance by `off(day)`
- 4. update field

- 1. dereference `today`
- 2. examine RTTI
- 3. determine `off(day)`
- 4. advance by `off(day)`
- 5. update field
Comments on use of RTTI in PLs

- When and how is \texttt{off(\texttt{day})}, the function determining the field offset, determined?

- In statically typed languages:
  - at compile time
  - from the static type of \texttt{today}.

- In dynamically typed languages:
  - at runtime
  - from the RTTI of “\texttt{*today}”

- In C, the \texttt{actual} type of \texttt{*today} could be \texttt{anything} (due to type punning).

- In Java, the \texttt{actual} type of the object that \texttt{today} refers to, can be any class that \texttt{extends} class \texttt{Date}. 