5 Storage

Contents [146 frames]

5.1 Storage models [21 frames] .......................... 1
  5.1.1 Utopic perspective of memory [14 frames] 2
  5.1.2 Real-world memory models [2 frames] .... 5
  5.1.3 Classical storage model [2 frames] ...... 5
  References ........................................ 5

5.2 Arrays [24 frames] ................................ 6
  5.2.1 Varieties of arrays [12 frames] ......... 6
  5.2.2 Arrays with integral index types [7 frames] 8
  5.2.3 Type of arrays [3 frames] .............. 9
  References ........................................ 9
  Exercises ........................................ 9

5.3 Variables’ life time [35 frames] ................. 10
  5.3.1 Simple lifetime [6 frames] .............. 10
  5.3.2 Storage class [10 frames] ............ 11
  5.3.3 The heap [9 frames] ................. 13
  5.3.4 Dangling references [6 frames] ...... 14
  5.3.5 Heap errors [4 frames] .............. 15
  References ........................................ 15

5.4 Representation of types in memory [22 frames] 16
  5.4.1 Simple representation of types in mem-
      ory [6 frames] ................................ 16
  5.4.2 Reference vs. value semantics [10 frames] 17
  5.4.3 Shared representation & lazy
      copy [10 frames] .............................. 19
  5.4.4 Value vs. reference semantics in various
      PLs [7 frames] ................................. 21
  References ........................................ 22
  Exercises ........................................ 22

5.5 Automatic memory management [16 frames] 22
  References ........................................ 25

5.6 Run time type information [16 frames] 25
  References ........................................ 28
  List of tables .................................... 28
  List of figures .................................. 29
  List of equations .............................. 29

References ........................................... 29

Exercise .............................................. 29

References ........................................... 29

List of tables ..................................... 28
List of figures .................................... 29
List of equations .............................. 29

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., per-
    son’s weight

1. Storage: a visual mindmap

Figure 5.0.1: Storage: a visual mindmap

2. Storage models: a visual mindmap

Figure 5.1.1: Storage models: a visual mindmap

3. Storage & the need for variables
Definition 5.1.1 (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

5.1.1 Utopic perspective of memory

14 Frames: 
- Store, cells & values
- Confusing terminology: variables vs. cells?
- Life cycle of a cell in the store
- Cells & references to cells in ML
- Retrieving cell contents in ML
- Mutable cells in ML
- More on mutable cells in ML
- Left vs. right occurrence of variable
- L-Value vs. R-value of an expression
- L-values vs. R-values in ML
- More on L-values in C
- Composite variables
- Selective/total inspect/update
- Total or selective update

5. 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

Figure 5.1.2: Store, cells & values

| war | 17 | ? | lord | 1984 |

Figure 5.1.3: Legend of Chapter 5

- Unspecified thingy
- Persistent cell
- Ephemeral cell
- Free Cell
- Allocated cell
- Allocated cell
- Uninitialized cell
- Cell with integer value
- Cell with string value
- Cell with composite Value

6. 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

7. 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

8. 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:

```
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:

```
val s = r;
val s = ref 3 : int ref
```

Figure 5.1.5: A cell and names of two references to it in ML

r and s are not cells; they are merely names of values.

9. Retreiving cell contents in ML

• get 3, the contents of this cell:

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

• !s is the same as !r:

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

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

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

Figure 5.1.6: Retrieving cell contents in ML

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

10. Mutable cells in ML

The ML memory model is close to the utopic

• change the value stored in this cell:

```
r := 2;
val it = () : unit
```

• get this value:

```
!r;
val it = 2 : int
```

• get the same value:

```
!s;
val it = 2 : int
```

Figure 5.1.7: Mutating a cell contents in ML

11. More on mutable cells in ML

• change the contents of this cell:

```
s := !r + 4;
val it = () : unit
```

• now get the content of the cell referenced by r:

```
!r;
val it = 6 : int
```

Figure 5.1.8: Reading & mutating a cell contents in ML

12. Left vs. right occurrence of variable

What’s the difference between the two occurrences of v?

```
Left vs. right
v ← ϕ(v,e₁,e₂,...,eₙ)
```

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. Evaluate v
2. Treat the result as reference to a cell.
3. Retrieve the contents of that cell
4. Use this contents as the value of v
5. You can forget about the cell now

13. **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

The expression \( *s++ \) has an L-value!

Function \( \text{min} \) returns an expression which has an L-value (just as an R-value).

14. **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

\[ \text{val } a = 19; \]

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

\[ \text{val } r := \text{ref 17}; \]

then:

- !\( r \) is the contents of this cell (R-value); it can be used e.g., by

\[ \text{val } x := \text{ref 17}; \]
- \( r \) is the reference to this cell (L-value); it can be used e.g., by

\[ r := x + 3; \]

15. **More on L-values in C**

Not every value is an L-value:

\[ 0 = 1; \quad // \quad \text{not an L-value} \]

Not every L-value is modifiable:

\[ \text{const int } i = 0; \]
\[ i = 1; \quad // \quad \text{unmodifiable L-value} \]

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

\[ \text{int max}(int & x, int & y) \{ \]
\[ \quad \text{return } x + y - \text{min}(x, y); \quad // \quad \text{implicit conversion} \]
\[ \}
\[ \text{max}(a, b) = \text{max}(c, d); \quad // \quad \text{X not an L-value} \]

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

\[ &1; \quad // \quad \text{not an L-value} \]
\[ \&i; \quad // \quad \text{an (unmodifiable) L-value} \]
\[ \&\text{max}(a,b); \quad // \quad \text{not an L-value} \]
\[ \&\text{min}(a,b); \quad // \quad \text{is an L-value!} \]

16. **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:

\[ \text{VAR} \]
\[ \text{today: Date;} \]

\( \text{today} \) access the entire value stored in this variable

\( \text{today.d} \) access a component of the value stored this variable

17. **Selective/total 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.

- 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).
### 18. Total or selective update

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
```

### 5.1.2 Real-world memory models

#### 19. Memory structure: the 8086 architecture

#### Segment Registers

- **DS**: Data Segment
- **CS**: Code Segment
- **SS**: Stack Segment
- **ES**: Extra Segment

#### Offset Registers

- **AX, BX, CX, DX**: General Purpose
- **SP**: Stack Pointer
- **BP**: Back Pointer
- **SI, DI**: Offset Registers

#### 20. 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)

### 5.1.3 Classical storage model

#### 21. Classical model of memory

#### Segments:

#### Permissions:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Code</td>
<td>X</td>
<td>X</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>X</td>
</tr>
<tr>
<td>Stack</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 5.1.1: Permissions of memory segments in an idealized storage model

#### 22. 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);
    return r;
}

c 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-2];
}
```
5.2 Arrays

Contents [24 frames]
- 5.2.1 Varieties of arrays [12 frames] ........ 6
- 5.2.2 Arrays with integral index types [7 frames] ... 8
- 5.2.3 Type of arrays [3 frames] ........ 9
- References ......................... 9
- Exercises .......................... 9

5.2.1 Varieties of arrays

12 Frames: □ Variety I/V: static arrays □ Variety II/V: Stack based arrays □ Variety III/V: Dynamic arrays □ Variety IV/V: flexible arrays □ Variety V/V: associative arrays □ The unbelievable power of associative arrays □ The AWK implicit loop □ Summary: determining the index set □ Arrays' efficiency □ Sophisticated data structures as part of PLs? □ The sad story of PASCAL's sets □ Dilemmas in language design

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

5.2.2 Arrays with integral index types


- size determined at runtime
- size may change after creation
- allocated on the heap segment

5.2.3 Type of arrays

3 Frames: □ Variety I/V: static arrays □ Variety II/V: Stack based arrays □ Variety III/V: Dynamic arrays

- size determined at runtime
- size may change after creation
- array may expand or shrink

References

Exercises

23. Array variables

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

Definition 5.2.1 (Array values). An array value is a mapping from a set of indices to a set of values.

Definition 5.2.2 (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.

Fortran  Integral Exponentiation

C/Java  Integral Exponentiation

Pascal  Map

24. Why only now?

- Array values are not very useful
- But... array variables become very useful...
  - Efficient mapping into memory with the classical storage models
  - Foundation for many algorithms
  - Foundation for many data structures

5.2.1 Varieties of arrays

12 Frames: □ Variety I/V: static arrays □ Variety II/V: Stack based arrays □ Variety III/V: Dynamic arrays □ Variety IV/V: flexible arrays □ Variety V/V: associative arrays □ The unbelievable power of associative arrays □ The AWK implicit loop □ Summary: determining the index set □ Arrays' efficiency □ Sophisticated data structures as part of PLs? □ The sad story of PASCAL's sets □ Dilemmas in language design

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

1 with slight variation due to the fact that arrays first index is 0 rather than 1
2 Recall that subrange is a type constructor in PASCAL
3 Did you see any arrays in ML?
29. Variety V/V: associative arrays

```php
$wives["Adam"] = "Eve";
$wives["Lamech"] = "Adah, and Zillah";
$wives["Abraham"] = "Sarah";
$wives["Isaac"] = "Rebecca";
$wives["Jacob"] = "Leah, and Rachel";
```

- index can be anything, typically strings.
- common in scripting PLs, e.g., AWK, JavaScript, PHP
- typically, implemented as a hash table

30. The unbelievable power of associative arrays

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

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

Explanation follows...

31. The AWK implicit loop

**Computing word frequencies in AWK**

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

```awk
#!/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\text{th}
  # word in the current line
}
```

32. 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.

33. Arrays’ efficiency

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

- including range-based arrays, as in PASCAL
- including true multi-dimensional 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.

34. Sophisticated data structures as part of PLs?

- Associative arrays are great!
  - We want more, ...
    - sets!
    - multi-sets!!
    - stacks and queues and trees!!!

35. 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
  - parallelization
36. 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.4

37. 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
    \[ a[i] \equiv *(a+i) \equiv *(i+a) \equiv i[a] \] (5.2.1)
  - Implicit in, e.g., Java, array access it translated to simple machine instructions

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

38. 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...

4 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.

39. Layout of multi-dimensional arrays

Two Main Strategies:
- Multi-Layered Memory Mapping:
  1. row-major
  2. column-major
- Multiple Dereferencing

40. 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:

\[
\text{offset}(A_{i,j}) = (i-1)m + (j-1)
\] (5.2.2)

![Figure 5.2.1: Row-major layout of 2D arrays (e.g., Pascal)]

\( A: \text{a } 4 \times 4 \text{ matrix} \)

\[ \begin{matrix}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33}
\end{matrix} \]

column-major layout

\[ \begin{matrix}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33}
\end{matrix} \]

41. 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:

\[
\text{offset}(A_{i,j}) = (j-1)n + (i-1)
\] (5.2.3)

![Figure 5.2.2: Column-major layout of 2D arrays (e.g., Fortran)]

\( A: \text{a } 4 \times 4 \text{ matrix} \)

\[ \begin{matrix}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33}
\end{matrix} \]

column-major layout

\[ \begin{matrix}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33}
\end{matrix} \]
Address of \( A_{i,j} \) where \( A \) is a matrix, is given by:

\[
\text{address}(A_{i,j}) = \text{dereference}(\text{address}(A) + i - 1 + j) \quad (5.2.4)
\]

**Fact 5.2.3** (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.

**Exercise 1.** Generalize the offset calculation formula of the layered-memory layout to PASCAL arrays.

**Exercise 2.** What’s the output of the following AWK program?

```awk
for (i in A) print i;
```

**Exercise 3.** It is known that a language \( \mathcal{L} \) uses the “layered-memory” layout and that \( \mathcal{L} \) does not check array bounds. Write a program in \( \mathcal{L} \) that prints “row” if \( \mathcal{L} \) implementation on the machine in which the program is run, uses the “row-major” layout, and “column” if \( \mathcal{L} \) uses “column-major” layout.

**References**

- Arrays
- Arrays
- Associative Arrays
- Automatic Variable
- Call Stack
- Column-Major Order
- Dynamic Arrays
- Row Major Order
- Stack-based Memory Allocation

**Exercises**

1. Generalize the offset calculation formula of the layered-memory layout to PASCAL arrays.
2. What’s the output of the following AWK program?

```awk
for (i in A) print i;
```

Why is the only sensible option? Repeat the above for JAVA.

3. It is known that a language \( \mathcal{L} \) uses the “layered-memory” layout and that \( \mathcal{L} \) does not check array bounds. Write a program in \( \mathcal{L} \) that prints “row” if \( \mathcal{L} \) implementation on the machine in which the program is run, uses the “row-major” layout, and “column” if \( \mathcal{L} \) uses “column-major” layout.
4. Explain why the “multiple-dereferencing” layout for representing multi-dimensional arrays is likely to be slower than than the “multi-layered memory mapping” layout.

5. Generalize the offset calculation formula of the layered-memory layout to three (four, five, six…) dimensions.


7. Explain how the “multiple-dereferencing” layout is applied in 3D arrays.

8. Lady D’Arbanville used to say that stack based arrays make the best of all worlds. Discuss her claim.

9. Explain how knowledge of whether the PL is row-based or column-based plays a role in the design of algorithms for large numerical problems.

10. Explain why the “multiple-dereferencing” layout for representing multi-dimensional arrays tends to be more memory efficient than “multi-layered memory mapping”.

11. Explain how it follows from the classical storage model that PASCAL functions are forbidden from returning arrays.

12. How does JavaScript support multi-dimensional arrays?

13. Name a data structure with a very efficient array implementation?

14. Demonstrate a case in which the “multiple-dereferencing” layout for representing multi-dimensional arrays is less efficient in its use of memory than the “multi-layered memory mapping” layout.

15. Explain how it follows from the classical storage model that PASCAL functions are forbidden from returning arrays.

16. Would you say that array variables more useful than array values? Discuss your position.

17. Where are “shebangs” used in this unit?

18. What are the pros and cons of the modification to the multiple-dereferencing layout, in which a reference to a zero length array is used instead of a null reference.

19. Which features does C++ offer in support of data structures library? and Java? and Pascal? and AWK? and Go?

20. Eitan Ha-Ezrahi claimed that he can extend Java to include associative arrays. Explain why this is not such a good idea.

21. Pascal does not allow to overflow array bounds. Write a program that uses timing to demonstrate that the language uses “row-major” layout. (Hint: locality of reference)

22. Are the following Pascal types equivalent?

23. Daniel Webster claimed that all integer arrays of C++ (Java, Pascal, AWK) are of the same type. Discuss his claim.

5.3 Variables’ life time

Contents [35 frames]

5.3.1 Simple lifetime [6 frames] ................. 10
5.3.2 Storage class [10 frames] ................. 11
5.3.3 The heap [9 frames] ..................... 13
5.3.4 Dangling references [6 frames] ........... 14
5.3.5 Heap errors [4 frames].................... 15

5.3.1 Simple lifetime

Definition 5.3.1 (Variable lifetime). The period between allocation of a certain variable and its deallocation is called the lifetime of that variable

Main varieties: 5

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

5.3.2 Persistent variable lifetime

Definition 5.3.2 (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.

5we will see more in the slides below
49. 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.

50. More on the simplistic approach

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:
  \[
  d_1 = d_2 \quad d_1 \subset d_2 \quad d_2 \subset d_1 \quad d_2 \cap d_1 = \emptyset
  \]

51. 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

52. 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

53. Terms “local” & “global” are confusing

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

**Definition 5.3.3** (Storage class in C/C++).

A storage-class specifier in C or C++, is one of the keywords \texttt{auto, register, static, extern, typedef}, or \texttt{thread_local};

- it is used mainly for specifying the lifetime of a variable and its scope.

54. Better terminology

Can be understood in terms of two of these keywords:

- **Block activation variable** designated by \texttt{auto}; allocated on the stack
- **Program activation variable** designated by \texttt{static}; allocated in the data segment.

55. Approximate meaning of C’s storage specifiers

- **auto**: block activation
  
  block variables with no storage-class specifier default to \texttt{auto}

- **register**: same as \texttt{auto}, but with recommendation to place in a register
- **static**: program activation
- **extern**: program activation

\(^6\)the next few slides will discuss these in greater detail
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

56. **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;
}
```

57. **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;
}
```

58. **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</td>
</tr>
<tr>
<td>register</td>
<td>✓</td>
<td>same as auto</td>
</tr>
<tr>
<td>static</td>
<td>✓</td>
<td>program activation</td>
</tr>
<tr>
<td>extern</td>
<td>✓</td>
<td>same as static</td>
</tr>
</tbody>
</table>

Table 5.3.1: C’s storage specifiers in blocks

59. **Examples: C’s storage specifiers at the external level**

**File a.c**

```c
auto x; // X
register double y; // X

/* 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)(); // X

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

int isPrimesArray[] = {
    ... /* Accessible from other files */
}
```

60. **C: access to entities defined in another file**

**File b.c**

```c
extern N = 100; // X
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[];
```

61. **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.

62. **C’s storage specifiers at the external (file) level**
5.3.3 The heap

**Table 5.3.2:** 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 static¹¹</td>
</tr>
<tr>
<td>auto</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>x</td>
<td>program activation</td>
</tr>
<tr>
<td>static</td>
<td>✓</td>
<td>same as static¹²</td>
</tr>
<tr>
<td>extern</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 5.3.4 (Heap variables).** *Heap variables are anonymous variables whose lifetime 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

5.3.3 The heap

**Definition 5.3.4** (Heap variables). *Heap variables are anonymous variables whose lifetime 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

**5.3.3 The heap**

**Table 5.3.2:** 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 static¹¹</td>
</tr>
<tr>
<td>auto</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>x</td>
<td>program activation</td>
</tr>
<tr>
<td>static</td>
<td>✓</td>
<td>same as static¹²</td>
</tr>
<tr>
<td>extern</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### 5.3.3 The heap

**Definition 5.3.4 (Heap variables).** *Heap variables are anonymous variables whose lifetime 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

**5.3.3 The heap**

**Definition 5.3.4** (Heap variables). *Heap variables are anonymous variables whose lifetime 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

**63. The heap**

**Definition 5.3.4 (Heap variables).** *Heap variables are anonymous variables whose lifetime 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

**64. 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”

**65. Motivation**

**Program garbage.** (* Truly useless program *).

```pascal
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.

### 5.3.3 The heap

**Definition 5.3.4 (Heap variables).** *Heap variables are anonymous variables whose lifetime 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

**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 `new()` (predefined)
- **C++’s** Operator `delete` (builtin; can be overloaded)
- **Java** Automatically, by the GC

**66. 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 `new()` (predefined)
- **C++’s** Operator `delete` (builtin; can be overloaded)
- **Java** Automatically, by the GC

**67. Linked list with heap variables**

**Pascal**

```pascal
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);
```

**Java**

**68. Access to heap variables**

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

**Definition 5.3.5 (Pure reference, referring, and dereferencing).**

- A pure reference or reference or short¹⁴ 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.

¹³GC = Garbage Collector

¹⁴not to be confused now with C++ references
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 “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, 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.

Dangling references

6 Frames: □ Dangling references □ How are dangling references created? □ Language protection against dangling references □ Quiz: What’s dangling here? □ What’s the penalty of accessing a dangling reference □ Stack corruption via dangling reference into an automatic variable

Definition 5.3.6 (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

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 *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 b = f();
b();
// May access a dangling reference
```

A new C++ keyword
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
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!

---

**Table 5.3.3: Language protection against dangling references**

<table>
<thead>
<tr>
<th>Freed memory</th>
<th>Stack reference</th>
<th>Inner functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>programmer’s responsibility</td>
<td>no inner functions</td>
</tr>
<tr>
<td>GNU-C</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>programmer’s responsibility</td>
<td>programmer’s responsibility</td>
</tr>
<tr>
<td>PASCA</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>programmer’s responsibility</td>
<td>cannot take address of stack variables</td>
</tr>
<tr>
<td>JAVA</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>programmer never deallocates memory, thanks to garbage collection</td>
<td>objects: always functions are not on the heap; first class; their stack: scalars or references taken to objects; no references to stack</td>
</tr>
</tbody>
</table>

---

**Quiz: What’s dangling here?**

```
#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!
"
        )
    );
    (void) printf("q=%p\n", q);
    (void) printf("q->a=%d\n", p->a);
    (void) printf("q->q=\%s\n", p->q);
    return 0;
}
```

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

---

### 5.3.5 Heap errors

#### Definition 5.3.7 (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
### 79. Memory leak

**Definition 5.3.8 (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;
}
```

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.

### 5.4 Representation of types in memory

#### Contents [33 frames]

- 5.4.1 Simple representation of types in memory [6 frames]
- 5.4.2 Reference vs. value semantics [10 frames]
- 5.4.3 Shared representation & lazy copy [10 frames]
- 5.4.4 Value vs. reference semantics in various PLs [7 frames]

#### 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 ⇒ Machine** how to represent values on different machines

**Machine ⇒ Program** how to represent values on different machines

#### 83. 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 $\mathcal{P}_\mathcal{L}$ for the representation of $\forall \mathcal{X}$ on machines $M_1, M_2, \ldots$:

$$
\mathcal{P}_\mathcal{L} : \mathcal{X} \rightarrow \{M_1, M_2, \ldots\}
$$

most often, employs the recursive nature of the type system

- How to represent atomic types?
- How to represent type constructors?
**Policy** $P_L$

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

sets a *contract* between two parties:

**Implementor of $L$** Given a machine $M_i$ select mapping $V_L \rightarrow M_i$ 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 $V_L \rightarrow M_i$

### 85. Exception I: non-primitive atomics

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

### 86. 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

### 87. 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.

**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.

### 84. Implicit contract of type representation policy

**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

### 88. Variables: reference vs. value semantics

**Value Semantics**

- Figures 5.4.1: Value semantics

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

**Reference Semantics**

- Figures 5.4.2: Reference semantics

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

### 89. 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

---

16 Can you detect and explain all the syntactical differences between the two languages?
90. 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

91. “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
r = v; // auto-unboxing
```

92. Boxing

Auto boxing: Coercion from, e.g., int to Integer

Auto unboxing: Coercion from, e.g., Integer to int

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

```java
Double dd = null;
double d = dd;
```

93. 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.,
96. 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(2015, 04, 21);
    Date tomorrow(2015, 04, 22);
    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(2015, 04, 21);
    Date tomorrow = new Date(2015, 04, 22);
    today = tomorrow;
    tomorrow.year = 3025;
    System.out.println(today.year);
}
```

97. Comparing the two semantics

C++: Value semantic

```cpp
// Creating values for today and tomorrow
Date today = Date(2015, 04, 21);
Date tomorrow = Date(2015, 04, 22);
// Assigning variable tomorrow to today
today = tomorrow;
today.year = 3025;
today.year = 3025;
cout << today.year;
```

Java: Reference semantic

```java
// Creating values for today and tomorrow
Date today = new Date(2015, 04, 21);
Date tomorrow = new Date(2015, 04, 22);
// Assigning variable tomorrow to today
today = tomorrow;
today.year = 3025;
System.out.println(today.year);
```

98. Which semantic does ML use?

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.

99. Efficient Lisp implementation with references

- Let α, β be two large S-expressions.
- (setq a α)
- (setq b β)
- (cons a b)
- (setq c (cons a b))

Figure 5.4.4: Sharing of representation in Lisp values

100. Generalization

Definition 5.4.1 (Lazy copying). Generalizing the 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 Lisp approach
- Support for languages which permits mutation of values.
Many extensions of pure-Lisp allow such mutation.

Mutation is the bread an butter of imperative programming.

Conceptually similar to “copy on write” in memory management.

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(2015, 04, 21);
Date *tomorrow = new Date(2015, 04, 22);
today = tomorrow; // Leak!
tomorrow->year = 3025;
cout << today->year;
delete tomorrow;
delete today; // Heap corruption?
```

Output is... 3025

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

```java
class Date implements Cloneable {
    :
}
Date today = new Date(2015, 04, 21);
Date tomorrow = new Date(2015, 04, 22);
today = (Date) tomorrow.clone();
tomorrow.year = 3025;
System.out.println(today.year);
```

Output is... 2015

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?

Reference assignment
Only the reference is copied

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

Shallow copy
Only the referenced value is copied

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

After assignment

Figure 5.4.6: Assignment strategies side by side

106. Assignment strategies side by side

107. 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 assignment</td>
<td>Never</td>
<td>X</td>
</tr>
<tr>
<td>Shallow copy</td>
<td>Maybe</td>
<td>X</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>

Table 5.4.1: Behind the scenes of four semantics of assignment

5.4.4 Value vs. reference semantics in various PLs

7 Frames: What does Java clone() do? Working knowledge of semantics? Overloading the assignment operator in C++ Case study: assignment & copy in Eiffel More general working knowledge Semantics in some contemporary PLs Overview: semantics of assignment

108. 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

18Typical exam question, if you like it phrased this way...

Deep clone

- if the class implements interface Cloneable, and
  - the programmer overrides the default clone method, and
  - correctly implements a “deep clone” semantic

109. 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.

110. Overloading the assignment operator in C++

And, what does the assignment operator do?19 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.

111. 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

19If you think you are smart, please repeat for copy constructor
Suppose that PASCAL had a list type constructor

```
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

### 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.\(^{21}\)

### Exercises
1. Which semantics is used in PASCAL? (explain why your answer could have been guessed)
2. Do function values use reference or value semantics?
3. Which semantics is used in GO?
4. Which assignment semantics is used in PASCAL?
5. Which assignment semantics is used in ML?
6. C offers automatically generated assignment operator; which semantics does this operator use?
7. C offers automatically generated constructor; which semantics does this constructor use?

### 5.5 Automatic memory management

16 Frames:
- Memory management
- Reference counting
- Pros & cons of reference counting
- What is garbage collection?
- Why garbage collection?
- Mark & sweep garbage collection
- Summary: mark & sweep garbage collection
- Delicate issues of the marking process
- Stop & copy garbage collection
- Defragmentation
- Predicaments of garbage collection
- Some responses to these predicaments
- Memory leak in garbage collection?
- Output of the above program
- Semantical memory leak\(^{22}\)
- GC & the stack: escape analysis

### Contents

| References | 25 |

\(^{21}\)as are some of the sweeping judgments made in this slide...

\(^{22}\)Note that the previous example exhausted memory for the sake of demonstration; it did not really create semantic garbage.
### 116. 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

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

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

**De-allocation** After each decrement, if RC(0) = 0: (i) de-allocate O; (ii) decrement RC for all children of O and, (iii) recursively de-allocate objects whose RC=0;

### 117. 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”

**Fact 5.5.1** (Write barrier). The formidable write barrier excludes the universal application of RC for memory management

### 118. What is garbage collection?

**Definition 5.5.2** (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.

---

[^23]: No other allocation command makes sense
[^24]: the amount of work that needs to be done in each memory write

### 119. 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.

### 120. Mark & sweep garbage collection

This is our storage bank, *which contains many “cells” 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” … 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*. 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*. Staring 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, and iterate, 1<sup>st</sup> cell for recycling, and iterate, 2<sup>nd</sup> cell for recycling, and iterate, and iterate, and iterate, and iterate, 3<sup>rd</sup> cell for recycling, and iterate, 4<sup>th</sup> 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

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

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

whose contents is:

```java
public class Leak {

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);
}
```
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 + " : \n            r.freeMemory());
    }
    l = cons(l);
}

128. 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)

129. Semantical memory leak

*Note that the previous example exhausted memory for the sake of demonstration; it did not really create semantic garbage.

Definition 5.5.3 (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); // Lots of GC activity
}

130. 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, local variables are:
  Stack allocated built-in, atomic types: int, double, boolean etc. (JAVA forbids
  Stack allocated References to classes and arrays.
  Heap allocated Classes and arrays (accessed only by references

5.6 Run time type information

16 Frames: ◊ The challenge of deep clone ◊ But, there is a catch…
    Memory = bits & bytes! ◊ Example: different interpretations of a single byte ◊ Example: different interpretations of a 16 bits word ◊ Example: different interpretations of 32 bits word ◊ The layout of a C structure ◊ Summary: the “meaning” of bits and bytes ◊ A step in a BFS/DFS tour ◊ But, the visited block could be of any type! ◊ Interpreting bit and bytes as values of a type ◊ Designing an algorithm for traversing values ◊ RTTI is the answer! ◊ C, C++, & RTTI ◊ Use of RTTI in the implementation of different PLs ◊ Comments on use of RTTI in PLs

Contents

References . . . . . . . . . . . . . . . . . . . . . . 28

131. 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 5.6.1 (Network Traversal: breadth- (or depth-) first search). In Each Value we Visit:

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

Definition 5.6.2 (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!

133. 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

134. Example: different interpretations of a single byte

Figure 5.6.2: An 8-bit byte in memory

135. Example: different interpretations of a 16 bits word

Figure 5.6.3: A 16 bits word in memory

136. 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;
};

Figure 5.6.4: Layout in memory of a 32-bits word used as µ-Lisp Cons record

137. 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.
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

**Definition 5.6.3** (Static typing). The compiler knows the “deciphering key”, and it generates code based on this information.

**Definition 5.6.4** (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!

**Figure 5.6.7: Reference representation of the RTTI tag attached to values**

138. Summary: the “meaning” of bits and bytes

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.

**Figure 5.6.6: A step in a BFS/DFS tour**

139. A step in a BFS/DFS tour

140. But, the visited block could be of any type!

141. Interpreting bit and bytes as values of a type

142. 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 the different that occur in it.
**Definition 5.6.5** (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

**144. 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-compatability 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.

**145. 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>syntax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static typing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic typing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTTI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type punning</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6.1: Use of RTTI in the implementation of different PLs

**146. Comments on use of RTTI in PLs**

- When and how is off(day), the function determining the field offset, determined?
- In statically typed languages:
  - at compile time
  - from the static type of today.
- In dynamically typed languages:
  - at runtime
  - from the RTTI of “*today”
- In C, the actual type of *today could be anything (due to type punning).
- In JAVA, the actual type of the object that today refers to, can be any class that extends class Date.

References

- Garbage Collection
- Variable
- Object Copy

List of tables

5.1.3 Classical storage model

5.1.1 Permissions of memory segments in an idealized storage model ............... 5

5.3.2 Storage class

5.3.1 C’s storage specifiers in blocks ............... 12
5.3.2 C’s storage specifiers at the external (file) level 13

5.3.4 Dangling references

5.3.3 Language protection against dangling references ............... 15

5.4.3 Shared representation & lazy copy

5.4.1 Behind the scenes of four semantics of assignment ............... 21
5.6.1 Use of RTTI in the implementation of different PLs ............... 28
List of figures

5.0.1 Storage: a visual mindmap ............... 1
5.1.1 Storage models: a visual mindmap .... 1

5.1 Utopic perspective of memory
5.1.1 Store, cells & values .................... 2
5.1.2 Legend of Chapter 5 ..................... 2
5.1.3 Life cycle of a cell in the store ........ 2
5.1.4 A cell and names of two references to it in ML 3
5.1.5 Retrieving cell contents in ML ........ 3
5.1.6 Mutating a cell contents in ML ........ 3
5.1.7 Reading & mutating a cell contents in ML 3
5.1.8 Real-world memory models ............ 5
5.1.9 Address computation in the 8086 hardware architecture ......................... 5
5.1.10 Extended vs. expanded memory in the ancient 8086 architecture ............... 5
5.1.11 Classical storage model .............. 5
5.1.12 C programs & the classical memory model .............. 5

5.2 Arrays with integral index types
5.2.1 Row-major layout of 2D arrays (e.g., Pascal) .......... 8
5.2.2 Column-major layout of 2D arrays (e.g., Fortran) ........ 8
5.2.3 "Multiple dereferencing" layout of 2D arrays .......... 9
5.2.4 Layout of triangular array with the multiple dereferencing layout .............. 9
5.3.3 The heap
5.3.1 Lists with shared representation ........ 13

5.4 Reference vs. value semantics
5.4.1 Value semantics ......................... 17
5.4.2 Reference semantics ..................... 17
5.4.3 References to objects .................... 18

5.4.3 Shared representation & lazy copy
5.4.4 Sharing of representation in Lisp values .......... 19
5.4.5 Semantics dilemma of assignment in reference semantics ......................... 20
5.4.6 Assignment strategies side by side .......... 21
5.5.1 Mindmap of memory management .......... 22
5.6.1 Dereferencing a pointer to an unfamiliar memory address .......... 26
5.6.2 An 8-bit byte in memory ................. 26
5.6.3 A 16-bits word in memory ............... 26
5.6.4 Layout in memory of a 32-bits word used as µ-LISP Cons record .......... 26
5.6.5 Different "legitimate" interpretations of a 16-bytes memory block .......... 26
5.6.6 A step in a BFS/DFS tour ............... 27
5.6.7 Reference representation of the RTTI tag attached to values .............. 27

List of equations

5.2.2 Arrays with integral index types

5.2.1 Array access in C is a matter of pointer arithmetic ........................................ 8
\[ a[i] \equiv *(a+i) \equiv *(i+a) \equiv i[a] \]

5.2.2 Offset computation in 2-dimensional arrays stored in column-major layout ................. 8
\[ \text{offset}(A_{i,j}) = (i-1)m + (j-1) \]

5.2.3 Offset computation in 2-dimensional arrays stored in column-major layout ................. 8
\[ \text{offset}(A_{i,j}) = (j-1)n + (i-1) \]

5.2.4 Address computation in 2-dimensional arrays with the "multiple dereference" method ............... 8
\[ \text{address}(A_{i,j}) = \text{dereference}(\text{address}(A) + i-1) + j \]

5.4.1 Simple representation of types in memory

5.4.1 PL policy for representation of types ........ 16