Variables

Six properties:

1. name
2. address
3. type
4. value
5. lifetime
6. scope

• Binding times of properties:
  – language specification time
  – language implementation time
  – compile time
  – run time

• Broad classification:
  – static
  – dynamic
1. Name

• Syntax of language specifies valid names.
• Languages used to limit length of names
  – Fortran77: 6 characters
  – Fortran95: 31 characters
  – Java, C#: no limit
• Restrictions:
  – keywords have special meaning (e.g. ‘for’)
  – reserved words cannot be used as names
  – generally keywords are reserved
2. Address

- The memory address at which the value associated with the name is stored.
- This is called the l-value (because it is the value used on the left hand side of an assignment operator). In Scheme terms, an unevaluated symbol.
- A given name can be associated with multiple addresses during program execution (e.g. parameters of a recursive function, an instance variable defined in a multiply-instantiated class).
- Aliasing occurs when many variables share an address.
3. Type

- The type of a variable determines the set of values which can be associated with the name.
- We will return to type binding and type checking.
Aside: binding times of types

• The binding time of a set of values to a type differs across languages.
• In Java the size and representation for types is written into the language specification.
  – every Java implementation is guaranteed to have the same range of values for the primitive types.
• In C the binding of a set of value to types is left to implementation time.
  – different implementations can (and do) make different decisions
  – implication is that the very same program can behave differently on different platforms.
Binding types to names (1)

• Explicit vs. implicit
  – an explicit declaration specifies the type of a variable
  – an implicit declaration is a language convention giving name-type mappings
    • Fortran: “If the identifier begins with one of the letters I, J, K, L, M or N, or their lowercase versions, it is implicitly declared to be Integer type, otherwise it is implicitly declared to be Real type.” [p. 213]
  – Perl: different namespaces for different types
    • $ implies scalar
    • @ implies array
    • % implies hash
Binding types to names (2)

• Static vs. dynamic
  – with static binding the type of a variable is known at compile time (e.g. C, C++, Java)
  – with dynamic binding the type of a variable is not known until run time (e.g. Scheme, JavaScript), and can change during the execution of a program [pg. 214]:
    ```
    list = [10.2, 3.5];
    list = 47;
    ```
Binding types to names (3)

- Type inferencing (ML)
- ML derives the types of all expressions from the types of their constituent parts.
- ML (with very few exceptions) does not require any type declarations, and yet all expressions are fully typed at compile time.
- ML uses type variables to express generic types (e.g. 'a and "a).
4. Value

• The r-value of the variable (because it is the value used on the right hand side of an assignment operator). In Scheme terms, the value retrieved from symbol lookup in an environment.

• This is the contents of a block of memory cells, whose starting address is the l-value of the variable, whose size and interpretation are determined by the type of the variable.

• Sometimes we refer to this block of memory, typically many bytes large, simply as the “memory cell” of a variable.
Aside: memory footprint & representation

- The type of a variable determines two very important things:
  - the amount of memory allocated to the variable
  - the representation scheme used to write/read bit patterns into/from memory.
- Consider the types `short` and `int` in C. Using the default ‘cc’ compiler on pollux, we find that a `short` occupies 2 bytes (16 bits); an `int` occupies 4 bytes (32 bites).
- Integers (of any length) are stored using the 2’s complement representation scheme.
Anatomy of a short

The range of values we can store in a short is -32768 to +32767:

<table>
<thead>
<tr>
<th>BIT PATTERN</th>
<th>DECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111111111111111</td>
<td>+32767</td>
</tr>
<tr>
<td>0111111111111110</td>
<td>+32766</td>
</tr>
<tr>
<td>. . . . . .</td>
<td></td>
</tr>
<tr>
<td>00000000000000010</td>
<td>+2</td>
</tr>
<tr>
<td>000000000000000001</td>
<td>+1</td>
</tr>
<tr>
<td>000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>1111111111111111</td>
<td>-1</td>
</tr>
<tr>
<td>1111111111111110</td>
<td>-2</td>
</tr>
<tr>
<td>. . . . . .</td>
<td></td>
</tr>
<tr>
<td>100000000000000001</td>
<td>-32767</td>
</tr>
<tr>
<td>100000000000000000</td>
<td>-32768</td>
</tr>
</tbody>
</table>
Anatomy of an int

The range of values we can store in an int is -2147483648 to +2147483647:

<table>
<thead>
<tr>
<th>BIT PATTERN</th>
<th>DECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111111111111111111111111111</td>
<td>+ 2147483647</td>
</tr>
<tr>
<td>01111111111111111111111111111110</td>
<td>+ 2147483646</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . . . . . .</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000000000010</td>
<td>+2</td>
</tr>
<tr>
<td>00000000000000000000000000000001</td>
<td>+1</td>
</tr>
<tr>
<td>00000000000000000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>11111111111111111111111111111111</td>
<td>-1</td>
</tr>
<tr>
<td>11111111111111111111111111111110</td>
<td>-2</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . . . . . .</td>
<td></td>
</tr>
<tr>
<td>100000000000000000000000000000001</td>
<td>-2147483647</td>
</tr>
<tr>
<td>100000000000000000000000000000000</td>
<td>-2147483648</td>
</tr>
</tbody>
</table>

. . . . . . . . . . . . . . . . . .   |                |
What happens here?

int i = 32768;
short s = i;
5. Lifetime

• “The **lifetime** of a variable is the time during which the variable is bound to a specific memory location.” [p. 219]

• “...the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.” [p. 219]

• Four categories
  – static
  – stack-dynamic
  – explicit heap-dynamic
  – implicit heap-dynamic
Memory organization

- **STATIC**
  - code & static data

- **STACK**
  - local data: invocation records

- **AVAILABLE MEMORY**

- **HEAP**
  - dynamic data
static variables

• Bound to a memory location prior to execution.
• No run-time allocation needs to occur: efficient.
• Persistent: variable persists throughout execution of a program.
```c
#include "stdio.h"

int counter() {
    static int k = 0;
    return ++k;
}

int main() {
    printf("Counter is %d \n",counter());
    printf("Counter is %d \n",counter());
    return 0;
}

/* OUTPUT IS:
   Counter is 1
   Counter is 2
*/
Notes about example

• static variable k is allocated space in the static segment
• allocation happens once
• k’s lifetime is the that of the entire program
• k’s value persists from call to call
stack dynamic variables

• “storage bindings are created when their declaration statements are elaborated, but whose types are statically bound” [p. 220]

• Allocated on the run-time stack
#include "stdio.h";

int counter() {
    int k = 0;
    return ++k;
}

int main() {
    printf("Counter is %d \n", counter());
    printf("Counter is %d \n", counter());
    return 0;
}

/* OUTPUT IS:
   Counter is 1
   Counter is 1
   Counter is 1
*/
Notes about example

• stack-dynamic variable k is allocated space in the stack segment
• allocation happens each time function is called
• k’s lifetime is the that of the function invocation
• k’s value does not persist from call to call: it is reinitialized on each function execution