Low-level languages

- What are low-level languages?
  - Expose machine details (e.g., raw memory)
  - Statically compiled and weakly typed
- Motivation
  - “Systems programming”
  - Predictability
  - Performance
- Problems
  - Easy to make mistakes / hard to debug
  - Lower programmer productivity

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

Lvalues and rvalues

What does an assignment do?

\[ x = y \]

Store at location of \( x \) … … the contents of \( y \).

Variables \( (x, y) \) are names for storage locations that can hold some contents, e.g., integers.

Lvalues and rvalues (cont.)

- All expressions denote an rvalue:
  - \( x, 42, \text{arr}[4], x+1, \text{obj}.f, \sin(2.5) \), …
- Some expressions only denote an rvalue, but no lvalue:
  - \( 42, x+1, \sin(2.5) \), …
  - Can not appear on left-hand side of assign
  - Don’t have a defined storage location
  - Not modifiable
Objects and pointers

- **Object** = has both *lvalue* and *rvalue*
  - Different definition from OOP (object-oriented programming): in OOP, an object is an instance of a class

- **Pointer** = reference to object, usually implemented by address
  - For example: when *rvalue of p* is the same as *lvalue of q*, then p is a pointer to q

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

Pointers to stack objects

```c
void swap(int* x, int* y) {
    int x = *x;  // type pointer to int
    *x = *y;    // use rvalue as lvalue
    *y = x;    // use lvalue as rvalue
}
```

- Motivation: Emulate call-by-reference using pointers and call-by-value

Dangling references

- **Dangling reference** = pointer whose lifetime ends after the lifetime of its target
  - Tough bugs:
    - Reading through a dangling reference yields undefined results
    - Writing through a dangling reference may corrupt unrelated objects

Pointers to heap objects

```c
struct list {
    int _car;
    struct list* _cdr;
};
struct list* cons(int car, struct list* cdr) {
    struct list* result = malloc(sizeof(struct list));
    result->_car = car;
    result->_cdr = cdr;
    return result;
}
```

- Motivation: gain most of the advantages of structured programming, while retaining most of the advantages of assembly language

Pointers to heap objects (cont.)

```c
struct list* x = cons(2, cons(5, cons(1, NULL)));
for (i = x; i != NULL; i = i->_cdr)
    printf("%d", i->_car);
```

- Motivation: gain most of the advantages of structured programming, while retaining most of the advantages of assembly language
Heap object deallocation

- C uses explicit memory management
  ```c
  while (NULL != x) {
    struct list* y = x;
    x = x->cdr;
    free(y);
  }
  ```
- Memory leak = malloc without free
  - Program may run out of memory
  - May lead to crash after hours of deployment
- Dangling reference: use (read/write) after free

C features for pointers thus far

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
<td><code>T</code> *p;</td>
</tr>
<tr>
<td>assignment</td>
<td><code>p = q</code></td>
</tr>
<tr>
<td>dereference</td>
<td><code>p; p-&gt;f</code></td>
</tr>
<tr>
<td>address-of</td>
<td>&amp;x</td>
</tr>
<tr>
<td>cast (as type <code>T</code>)</td>
<td><code>(T*)x</code></td>
</tr>
<tr>
<td>size of type</td>
<td><code>sizeof(T)</code></td>
</tr>
<tr>
<td>size of variable</td>
<td><code>sizeof(*x)</code></td>
</tr>
<tr>
<td>allocation</td>
<td>malloc; free</td>
</tr>
<tr>
<td>pointer to nothing</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Address-of and dereference cancel each other out: `&(*x) == x`

Outline

- Concepts
- Uses of pointers
- Arrays
  - Pointer arithmetic
  - Multidimensional arrays

Arrays as pointers

- Subscript operator is syntactic sugar
  
  - `x[i] == *(x+i) == [x]`
  
- `&` and `*` (address-of and dereference) cancel each other out
  
  - `&e == e`
  
  - `&x[i] == &(*(x+i)) == x+i`
- What is the rvalue of `x`?
  
  - `x == x[0] == *(x+0) == &x[0]`
  
  - The rvalue of `x` is the lvalue of `x[0]`

Arrays as pointers (cont.)

- What does the `*` operator do for pointers?
  
  - `(uint)(x+i)`
  
  - `(uint)(&x[i])`
  
  - `(uint)(&x[0]) + sizeof(int) * i`
  
  - `(uint)x + sizeof(int) * i`
- In general:
  
  - `(uint)(p+i)`
  
  - `(uint)p + sizeof(baseType(p))/i`
C pointer features: addendum

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
<td>T* p</td>
</tr>
<tr>
<td>assignment</td>
<td>p = q</td>
</tr>
<tr>
<td>dereference</td>
<td>*p = f</td>
</tr>
<tr>
<td>address-of</td>
<td>&amp;x</td>
</tr>
<tr>
<td>cast</td>
<td>(T*)x</td>
</tr>
<tr>
<td>size of type</td>
<td>sizeof(T)</td>
</tr>
<tr>
<td>size of variable</td>
<td>sizeof(*x)</td>
</tr>
<tr>
<td>allocation, deallocation</td>
<td>malloc, free</td>
</tr>
<tr>
<td>pointer to nothing</td>
<td>NULL</td>
</tr>
<tr>
<td>identity comparison</td>
<td>==</td>
</tr>
<tr>
<td>subscript</td>
<td>i</td>
</tr>
<tr>
<td>arithmetic</td>
<td>+, -</td>
</tr>
<tr>
<td>magnitude comparison</td>
<td>&lt;, &lt;, =, &gt;=, &gt;</td>
</tr>
</tbody>
</table>

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

Pointer arithmetic

- Assume declarations "T* p1; T* p2; int i;"
- ptr + int → ptr
  - add multiple of size of pointed-to object
    - p1 + i == (T*)( (uint)p1 + i * sizeof(T) )
- ptr – int → ptr
  - subtract multiple of size of pointed-to object
    - p1 – i == (T*)( (uint)p1 – i * sizeof(T) )
- ptr – ptr → int
  - return difference in number of objects
    - p1 – p2 = ((uint)p1 – (uint)p2) / sizeof(T)

C strings

- C string = null-terminated array of char
  - char* x = "hello";
  - char* i;
  - for (i=x; \0 != *i; i++)
    - printf("%c\n", *i);
- The above code uses pointer arithmetic, would be clearer with array subscripts

C pointers illustrate usefulness of PL knowledge

- Emulating features in a language that does not directly support them
  - Call-by-reference
  - Recursive data types
- Reading obscure code
  - Treating arrays as pointers
  - Loops with pointer arithmetic

Arrays vs. pointers

- Arrays and pointers are almost the same &("x[i]") == x+i
- But it still matters how you declare them

<table>
<thead>
<tr>
<th>Declaration</th>
<th>int x[5];</th>
<th>int* x:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocates</td>
<td>5 integers</td>
<td>1 pointer</td>
</tr>
<tr>
<td>sizeof(x)</td>
<td>5 * sizeof(int)</td>
<td>sizeof(int*)</td>
</tr>
<tr>
<td>Has value?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>int x[i]: size determined by initialization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multidimensional arrays

<table>
<thead>
<tr>
<th>Declaration</th>
<th>int x[5][5];</th>
<th>int** x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocates</td>
<td>25 integers</td>
<td>1 pointer</td>
</tr>
<tr>
<td>sizeof(x)</td>
<td>25 * sizeof(int)</td>
<td>sizeof(int**)</td>
</tr>
<tr>
<td>Has lvalue?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- `int[5][5]` is array with contiguous layout
- `int**` may point to array with contiguous layout, or to row-pointer (array of arrays)

Outline

- Concepts
- Uses of pointers
- Arrays
- Pointer arithmetic
- Multidimensional arrays

How to read C definitions

- Rule of thumb:
  - Start at innermost identifier
  - Look right until ")" or end
  - Look left until "(" or start
  - Pop out one layer of "(...)" at a time

- Examples:
  - `int (*x)[3]` Pointer to array of 3 int.
  - `int (*x)[n]` Pointer to function returning int.
  - `int (*t)(x)[n]` Pointer to function returning array of unspecified number of pointers to int.

Multidimensional arrays

Scott Fig. 7.10: “char days[][10]” vs. “char* days[]”

- Contiguous array allocation
- True 2-dimensional array
- 7 * 10 * sizeof(char)

- Row pointer
- Array of pointers to arrays
- 7 * sizeof(char*) + 57 * sizeof(char)

Pointer arithmetic (cont.)

- `int x[5][5]`
  - Since x is array: x == &x[0]
  - Since x[i] is array: x[i] == &x[i][0]
- Assume (uint)x == 200, sizeof(int) == 4, what is the rvalue for each expression?
  - sizeof(*x), *x, *x+2, *x+2
  - x[2]+3, (x+2)->x, x[2]-x[0], x=x, x[2]=x[1]

Discussion

- Low-level languages are good for:
  - Device drivers
  - Memory managers
  - Network protocols
  - …
- Low-level languages have problems:
  - Less error checking
  - More verbose code
  - Missing high-level features