Outline

Scalar and Composite Types

Arrays

Records and Variants
Outline

Scalar and Composite Types

Arrays

Records and Variants
Machine Types as Scalars

- **Integer Types**
  - 8-bit (e.g. byte/char)
  - 16-bit (e.g. short)
  - 32-bit (e.g. int, long)
  - 64-bit (e.g. int, long, long long)
  - Exactly which C types are 64-bit specified by OS (ILP64, LP64, LLP64)

- **Addresses**
  - Pointers (next class)

- **Floating-point numbers** (usually IEEE 754 standard)
  - Half-precision (16-bit)
  - Single-precision (32-bit)
  - Double-precision (64-bit)
Non-machine Types as Scalars

- Multibyte Characters
  - e.g. Unicode
  - Usually mapped to same size integer type
- “Bigints”
  - Integers much bigger than machine word size
  - e.g. Python “longs”
- Multiprecision Arithmetic
  - Exact arithmetic with reals
  - e.g. numbers in Unix `bc`
Exotic Scalar Machine Types

- Decimal Numbers
  - Binary coded decimal
- What other types does your machine support?
Composite Types

- Arrays
  - Multiple elements, same type
- Records
  - Multiple elements, different types
- Variants/Unions
  - Multiple elements, different types
  - But one type at a time!
- Bitsets
- Sets, Tuples, Dictionaries/Maps
  - not covered today
Outline

Scalar and Composite Types

Arrays

Records and Variants
Array Concepts

- Usually contains multiple elements of the same type
- Can support multi-dimensional data
  - e.g. Matrix (2-D)
- Only array has name, elements are exclusively numerically indexed
  - Unlike maps/dictionaries
- Some languages provide additional features
  - e.g. consecutive elements of an array are also consecutive in memory
User Interface, Syntax and other Design Considerations

• [] or () for subscripts?
  • a[2] or a(2)?
  • a[1][2] or a[1, 2]?
  • C’s syntax also allows 3[a]?

• Allow access to “slices” of array?
  • Entire row (e.g. a[1][:], or entire column?
  • 2-D subset, n-D subset?
  • Are these also arrays?

• Extend scalar operations to arrays?
  • i.e. a + b is a[i] + b[i] when a and b are arrays?
  • could utilize SIMD units

• Should compiler/runtime track array “shape” or should user?
  • “shape” is size in each dimension
  • e.g. for a[2][5]; size(a, 1) is 2.
int a[5];

[ | | | | | ]
  0 1 2 3 4
  + + + + +
  a a a a a a

(Read as a + 0, a + 1, etc.)
Multidimensional arrays need to be mapped to 1-D memory

- e.g. if \(a[2][5]\), then accessing \(a[1][4]\) maps to 9

Multiple ways to map array subscripts to addresses

- Scheme 1: Row Major
  - Elements in same row are consecutive in memory

Most common format, used in C
- Scheme 2: Column Major
  - Elements in same column are consecutive in memory
- Used mostly by FORTRAN
• Scheme 3: Pointer to Arrays
  
  • Elements in outer dimensions are pointers to arrays in inner dimensions

• Only multidimensional array type in Java

• Also supported in C

```
char a[7][10];

char *b[7] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday",
    "Saturday"};
```
Layout of Pointer to Arrays
• Z-order Curve
• Hilbert Curves
Row Major
  - Array size a[2][5], base address is a
  - Access a[1][3] is a + 1 * 5 + 3
  - i.e. row * ncols + col

Column Major
  - Array size a[2][5], base address is a
  - Access a[1][3] is a + 3 * 2 + 1
  - i.e. col * nrows + row

In both cases, largely $O(1)$ for a fixed array for all indices
Compiler can also simplify calculations at compile-time if dimensions known
• Multidimensional arrays are arrays of *pointers*

• Array size *a[10]*, base address is *a*
  • Access *a[1][2]* is *deref(a + 1) + 2*

• Indirect addressing, requires reads of memory
• Z-Curve Ordering
  • ?
• Hilbert Ordering
  • ?
• More computation, but better locality
  • Is the tradeoff worth it?
Bounds Checking

• Should compiler track array “shape”?
• Should compiler throw runtime errors when out-of-bounds access occur?
  • e.g. access a[5] in int a[5]?

```c
int A[N];
for(int i = 0; i <= N; i++)
```
• What happens when elements beyond array length are written?

```c
int main(void) {
    int a[3];
    int N = 7;
    int i;

    for(i = 0; i < N; i++) {
        a[i] = 0xaabbccdd;
        printf("%d: %x\n", i, a[i]);
    }
}
```
(gdb) run
Starting program: /tmp/a.out
0: aabbccddd
1: aabbccddd
2: aabbccddd
3: aabbccddd
4: aabbccddd
5: aabbccddd
6: aabbccddd

Program received signal SIGSEGV, Segmentation fault.
0x00007ffffaabbccddd in ?? ()
Dynamic Arrays, Resizing Arrays

- Should the language support arrays whose size will be known only at runtime?
  - Stored on stack?
  - Stored on heap?
- Multidimensional dynamic arrays?

```c
void fn(int N) {
    int x[N];
    int *y = malloc(N*sizeof(int));
    ...
}
```

- Should resizing arrays be allowed?
Are strings arrays?
  - char a[] = "hello world";

Recall C does not store size of arrays
  - C uses ’\0’ as end-of-string marker.

What about Unicode? (or “wide characters”?)
  - These are multibyte character sets
How C-like languages deal with Unicode

- Recall '\0' means end of string in C
- Unicode is usually 16-bit, but can be up to 32 bits
- UTF-8 encoding:
  - 0–127 use a single byte
  - 128+ use multiple bytes (up to 4)
- How does indexing change?
Scalar and Composite Types

Arrays

Records and Variants
Also known as “product” types

```c
struct {
    char name[2];
    int atomic_number;
    double atomic_weight;
    char metallic;
}
```

What is the size of this struct if:

- char is 1 byte
- int is 4 bytes
- double is 8 bytes
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>atomic_weight</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td></td>
</tr>
<tr>
<td>metallic</td>
<td></td>
</tr>
</tbody>
</table>

4 bytes/32 bits
Record Layout

4 bytes/32 bits

| name | atomic_number | atomic_weight | metallic |
### Record Layout (Rearranged)

<table>
<thead>
<tr>
<th>name</th>
<th>metallic</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_number</td>
<td></td>
</tr>
<tr>
<td>atomic_weight</td>
<td></td>
</tr>
</tbody>
</table>

*4 bytes/32 bits*
Variants

- Like records, except all fields occupy same space

```c
union {
    int seconds_since_epoch;
    int julian_day;
} relative_date;
```

- The size of `relative_date` is size of `int`, not $2 \times int$
void print_relative_date(union relative_date x) {
    printf("Seconds since epoch: %d\n", 
           x.seconds_since_epoch);
}
struct relative_date_tagged {
    int is_seconds;
    union {
        int seconds_since_epoch;
        int julian_day;
    } relative_date;
}
Variant Use

- Should language check variants are being used correctly?
  - Initialization
  - Every use

```c
struct relative_date_tagged t;

if(t.is_seconds) {
    printf("Seconds since epoch: %d\n",
           t.relative_date.seconds_since_epoch);
} // code to handle julian days is missing!
```

- Some languages (notably the ML family) do not allow access to union members except in a separate match construct