Lecture 12

CIS 341: COMPILERS

Announcements

- Project 3: Compiling Control Flow
  - Due: Monday, February 25th at 11:59pm
- Midterm Exam:
  - Thursday, February 28th
  - In class
  - Examples on the web

STRUCTURED DATA

Compiling Structured Data

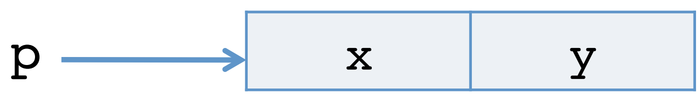
- Consider C-style structures like those below.
- How do we represent `Point` and `Rect` values?

```
struct Point { int x; int y; };

struct Rect  { struct Point ll, lr, ul, ur };

struct Rect mk_square(struct Point ll, int len) {
  struct Rect square;
  square.ll = square.lr = square.ul = square.ur = ll;
  square.lr.x += len;
  square.ul.y += len;
  square.ur.x += len;
  square.ur.y += len;
  return square;
}
```

Representing Structs

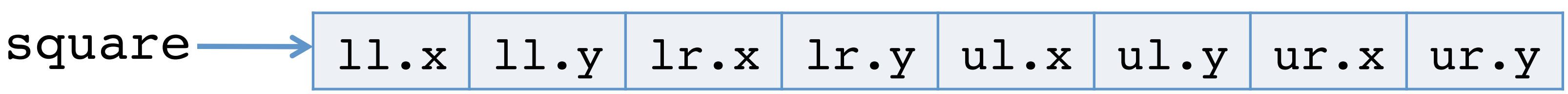
```
struct Point { int x; int y;};
```

- Store the data using two contiguous words of memory.
- Represent a `Point` value `p` as the address of the first word.




```
struct Rect  { struct Point ll, lr, ul, ur };
```

- Store the data using 8 contiguous words of memory.



- Compiler needs to know the *size* of the struct at compile time to allocate the needed storage space.
- Compiler needs to know the *shape* of the struct at compile time to index into the structure.

Assembly-level Member Access

square →

`ll.x`	`ll.y`	`lr.x`	`lr.y`	`ul.x`	`ul.y`	`ur.x`	`ur.y`

```
struct Point { int x; int y; };

struct Rect  { struct Point ll, lr, ul, ur };
```

- Consider: $C \vdash$ ⟦`square.ul.y`⟧ = (x86.operand, x86.insns)

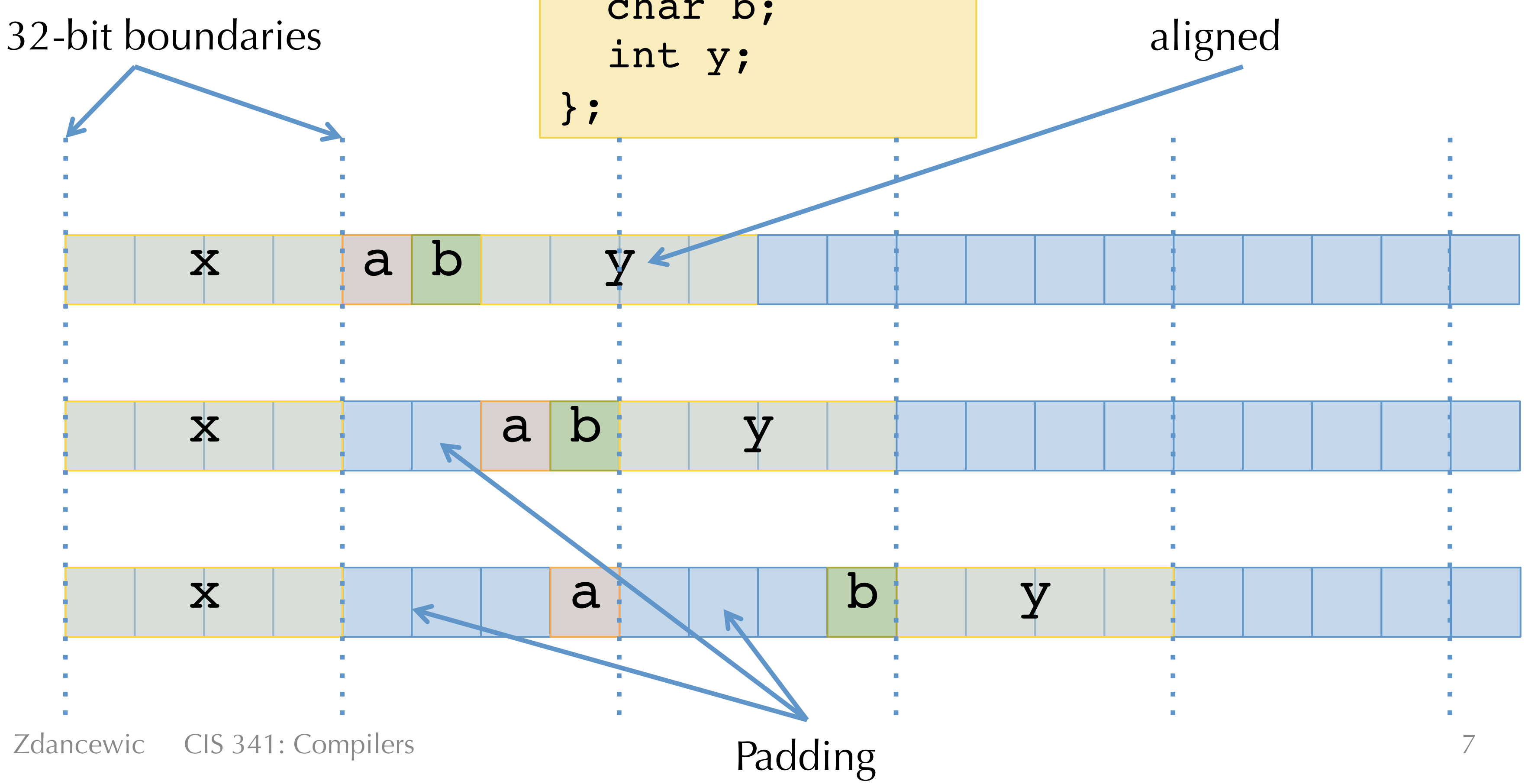
- Assume that ECX holds the base address of `square`
- Calculate the offset relative to the base pointer of the data:
  - `ul` = sizeof(struct Point) + sizeof(struct Point)
  - `y` = sizeof(int)

- So: ⟦`square.ul.y`⟧ = (ans, `Mov ans [ECX + 20]`)

Padding & Alignment

- How to lay out non-homogeneous structured data?

```
struct Example {
  int x;
  char a;
  char b;
  int y;
};
```



Copy-in/Copy-out

When we do an assignment in C as in:

```
struct Rect mk_square(struct Point ll, int elen) {
  struct Square res;
  res.lr = ll;
   ...
```

then we copy all of the elements out of the source and put them in the target. Same as doing word-level operations:

```
struct Rect mk_square(struct Point ll, int elen) {
  struct Square res;
  res.lr.x = ll.x;
  res.lr.y = ll.x;
  ...
```

- For really large copies, the compiler uses something like **`memcpy`** (which is implemented using a loop in assembly).

Procedure Calls

- Similarly, when we call a procedure, we copy arguments in, and copy results out.
  - Caller sets aside extra space in its frame to store results that are bigger than will fit in EAX.
  - We do the same with scalar values such as integers or doubles.
- Sometimes, this is termed "call-by-value".
  - This is bad terminology.
  - Copy-in/copy-out is more accurate.
- Problem: expensive for large records…
- In C: pass *pointers* to structs: "call-by-reference"
- Languages like Java and OCaml always pass non-word-sized objects by reference.

Call-by-Reference:

```
void mkSquare(struct Point *ll, int elen,
              struct Rect *res) {
  res->lr = res->ul = res->ur = res->ll = *ll;
  res->lr.x += elen;
  res->ur.x += elen;
  res->ur.y += elen;
  res->ul.y += elen;
}

void foo() {
  struct Point origin = {0,0};
  struct Square unit_sq;
  mkSquare(&origin, 1, &unit_sq);
}
```

- The caller passes in the address of the point and the address of the result (1 word each).
- Note that returning references to stack-allocated data can cause problems.
  - Need to allocate storage in the heap…

ARRAYS

Arrays

```
void foo() {
  char buf[27];

  buf[0] = 'a';
  buf[1] = 'b';
  ...
  buf[25] = 'z';
  buf[26] = 0;
}
```

```
void foo() {
  char buf[27];

  *(buf) = 'a';
  *(buf+1) = 'b';
   ...
  *(buf+25) = 'z';
  *(buf+26) = 0;
}
```

- Space is allocated on the stack for buf.
  - Note, without the ability to allocated stack space dynamically (C's `alloca` function) need to know size of buf at compile time…
- `buf[i]` is really just: (base_of_array) + i * elt_size

Multi-Dimensional Arrays

- In C, **`int M[4][3]`** yields an array with 4 rows and 3 columns.
- Laid out in *row-major* order:

`M[0][0]`	`M[0][1]`	`M[0][2]`	`M[1][0]`	`M[1][1]`	`M[1][2]`	`M[2][0]`	…

- M[i][j] compiles to?

- In Fortran, arrays are laid out in *column major order*.

`M[0][0]`	`M[1][0]`	`M[2][0]`	`M[3][0]`	`M[0][1]`	`M[1][1]`	`M[2][1]`	…

- In ML and Java, there are no multi-dimensional arrays:
  – (int array) array is represented as an array of pointers to arrays of ints.
- Why is knowing these memory layout strategies important?

Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they're in bounds.
  - Compiler generates code to test that the computed offset is legal
- Needs to know the size of the array… where to store it?
  - One answer: Store the size *before* the array contents.

`arr`

`Size=7`	`A[0]`	`A[1]`	`A[2]`	`A[3]`	`A[4]`	`A[5]`	`A[6]`

- Other possibilities:
  - Pascal: only permit statically known array sizes (very unwieldy in practice)
  - What about multi-dimensional arrays?

Array Bounds Checks (Implementation)

- Example: Assume EAX holds the base pointer (arr) and ECX holds the array index i. To read a value from the array `arr[i]`:

```
    Mov EDX  [EAX - 4]              // load size into EDX
    Cmp ECX  EDX                    // compare index to bound
    J l __ok                        // jump if  0 <= i < size
    Call __err_oob                  // test failed, call the error handler
__ok:
    Mov dest [EAX + 4*ECX]      // do the load from the array access
```

- Clearly more expensive: adds move, comparison & jump
  - More memory traffic
  - Hardware can improve performance: executing instructions in parallel, branch prediction
- These overheads are particularly bad in an inner loop
- Compiler optimizations can help remove the overhead
  - e.g. In a for loop, if bound on index is known, only do the test once

C-style Strings

- A string constant `"foo"` is represented as global data:

```
_string42: 102 111 111 0
```

- C uses null-terminated strings
- Strings are usually placed in the *text* segment so they are read only.
  - allows all copies of the same string to be shared.

- Rookie mistake (in C): write to a string constant.

```
char *p = "foo”;
p[0] = 'b’;
```

- Instead, must allocate space on the heap:

```
char *p = (char *)malloc(4 * sizeof(char));
strncpy(p, “foo”, 4);    /* include the null byte */
p[0] = 'b’;
```

TAGGED DATATYPES

C-style Enumerations / ML-style datatypes

- In C:

```
enum Day {sun, mon, tue, wed, thu, fri, sat} today;
```

- In ML:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Associate an integer *tag* with each case: `sun` = 0, `mon` = 1, …
  - C lets programmers choose the tags

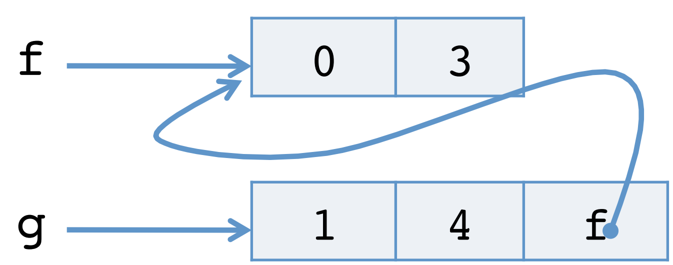
- ML datatypes can also carry data:

```
type foo = Bar of int | Baz of int * foo
```

- Representation: a `foo` value is a pointer to a pair: (tag, data)
- Example: tag(`Bar`) = 0, tag(`Baz`) = 1

  ⟦`let f = Bar(3)`⟧ =

  ⟦`let g = Baz(4, f)`⟧ =



Switch Compilation

- Consider the C statement:

```
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    …
    case sat: s3; break;
}
```

- How to compile this?
  - What happens if some of the break statements are omitted? (Control falls through to the next branch.)

Cascading `if`s and Jumps

⟦`switch(e) {case tag1: s1; case tag2 s2; …}`⟧ =

- Each `$tag1…$tagN` is just a constant int tag value.
- Note: ⟦break;⟧ (within the switch branches) is: `br %merge`

```
    %tag = ⟦e⟧;
    br label %l1
l1: %cmp1 = icmp eq %tag, $tag1
    br %cmp1 label %b1, label %merge
b1: ⟦s1⟧
    br label %l2

l2: %cmp2 = icmp eq %tag, $tag2
    br %cmp2 label %b2, label %merge
b2: ⟦s2⟧
    br label %l3
…
lN: %cmpN = icmp eq %tag, $tagN
    br %cmpN label %bN, label %merge
bN: ⟦sN⟧
    br label %merge

merge:
```

Alternatives for Switch Compilation

- Nested if-then-else works OK in practice if # of branches is small
  - (e.g. < 16 or so).
- For more branches, use better datastructures to organize the jumps:
  - Create a table of pairs (v1, branch_label) and loop through
  - Or, do binary search rather than linear search
  - Or, use a hash table rather than binary search

- One common case: the tags are dense in some range [min…max]
  - Let N = max – min
  - Create a branch table Branches[N] where Branches[i] = branch_label for tag i.
  - Compute tag = ⟦e⟧ and then do an *indirect jump*: J Branches[tag]
- Common to use heuristics to combine these techniques.

ML-style Pattern Matching

- ML-style match statements are like C's switch statements except:
  - Patterns can bind variables
  - Patterns can nest

```
match e with
| Bar(z) -> e1
| Baz(y, Bar(w)) -> e2
| _ -> e3
```

```
match e with
| Bar(z) -> e1
| Baz(y, tmp) ->
     (match tmp with
          | Bar(w) -> e2
          | Baz(_, _) -> e3)
```

- Compilation strategy:
  - "Flatten" nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from ⟦e⟧ to the variables bound in the patterns.
- There are many opportunities for optimization, many papers about "pattern-match compilation"
  - Many of these transformations can be done at the AST level