

COS320: Compiling Techniques

Zak Kincaid

January 29, 2026

Compiling data types

Structures

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

```
struct Rect { struct Point tl, br; };
```

```
struct Rect mk_square(struct Point top_left, long len) {  
    struct Rect square;  
    square.tl = top_left;  
    square.br.x = top_left.x + len;  
    square.br.y = top_left.y - len;  
    return square;  
}
```

How do we compile these structures?

```
struct Rect mk_square(struct Point top_left, long len)
```

- X86-64 calling convention:
 - Parameter 1 in `rdi`
 - Parameter 2 in `rsi`
 - Return in `rax`

```
struct Rect mk_square(struct Point top_left, long len)
```

- X86-64 calling convention:
 - Parameter 1 in rdi
 - Parameter 2 in rsi
 - Return in rax
- **Problem:** Parameter 1 doesn't fit into rdi, and return doesn't fit into rax

```
struct Rect mk_square(struct Point top_left, long len)
```

- X86-64 calling convention:
 - Parameter 1 in rdi
 - Parameter 2 in rsi
 - Return in rax
- **Problem:** Parameter 1 doesn't fit into rdi, and return doesn't fit into rax
- Straightforward solution: pass & return pointers to values that don't fit into registers (Java, OCaml)

```
struct Rect mk_square(struct Point top_left, long len)
```

- X86-64 calling convention:
 - Parameter 1 in `rdi`
 - Parameter 2 in `rsi`
 - Return in `rax`
- **Problem:** Parameter 1 doesn't fit into `rdi`, and return doesn't fit into `rax`
- Straightforward solution: pass & return pointers to values that don't fit into registers (Java, OCaml)
- C has copy-in/copy-out semantics ("call by value")
 - If we call `mk_square(p, 5)` and `mk_square` writes to `top_left.x`, the value of `p.x` does not change from the perspective of the caller

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- Solution for return:

```
struct Rect* mk_square(long top_left_x, long top_left_y, long len) {  
    struct Rect square;  
    ...  
    return &square;  
}
```

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- Solution for return:

```
struct Rect* mk_square(long top_left_x, long top_left_y, long len) {  
    struct Rect square;  
    ...  
    return &square;  
}
```

- Unsafe!

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- Solution for return:

```
struct Rect* mk_square(long top_left_x, long top_left_y, long len) {  
    struct Rect *result = malloc(sizeof(struct Rect));  
    ...  
    return result;  
}
```

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- Solution for return:

```
struct Rect* mk_square(long top_left_x, long top_left_y, long len) {  
    struct Rect *result = malloc(sizeof(struct Rect));  
    ...  
    return result;  
}
```

- Protocol: caller must de-allocate space
- *But* heap allocation is slow. Can we do better?

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- *Better* (and standard) solution for return:

```
void mk_square(struct Rect *result,  
              long top_left_x, long top_left_x, long len) {  
    ...  
    return;  
}
```

- Callee is responsible for allocating space for return value

Copy-in/Copy-out

- Solution: use additional parameters for structs

```
struct Rect mk_square(long top_left_x, long top_left_y, long len)
```

- *Better* (and standard) solution for return:

```
void mk_square(struct Rect *result,  
              long top_left_x, long top_left_x, long len) {  
    ...  
    return;  
}
```

- Callee is responsible for allocating space for return value

Structures in memory

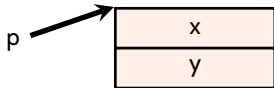
- What *is* a pointer to a structure?

Structures in memory

- What *is* a pointer to a structure?
- Address of the start of a block of memory large enough to store the struct

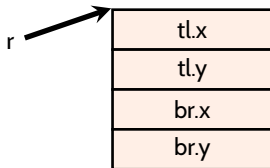
```
struct Point { long x, y; };
```

```
struct Point* p = malloc(sizeof(struct Point));
```



Structures in memory

- What *is* a pointer to a structure?
 - Address of the start of a block of memory large enough to store the struct
 - Nested structs:
`struct Rect { struct Point tl, br; };
struct Rect* r = malloc(sizeof(struct Rect));`

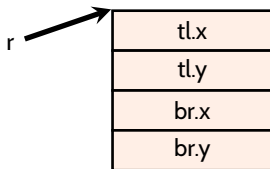


Structures in memory

- What *is* a pointer to a structure?
 - Address of the start of a block of memory large enough to store the struct

- Nested structs:

```
struct Rect { struct Point tl, br; };  
struct Rect* r = malloc(sizeof(struct Rect));
```



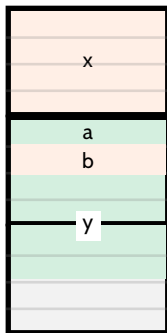
- Compiler needs to know:
 - **Size** of the struct so that it can allocate storage
 - **Shape** of the struct so that it can index into the structure

Padding & Alignment

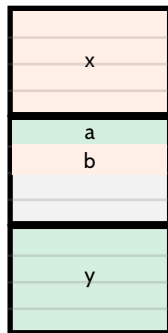
- Memory accesses need to be aligned
 - E.g., in x86lite, memory addresses are divisible by 8
 - Need to insert *padding*: unused space so that pointers align with addressable boundaries
- How do we lay out storage?

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

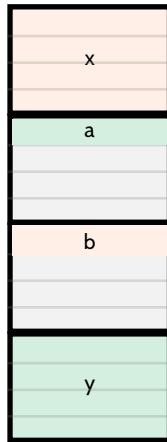
Note: 32-bit architecture



llvm packed



llvm unpacked



easy

Structures in LLVM

```
%Point = type { i64, i64 }
%Rect = type { %Point, %Point }

define void @mk_square(%Rect* noalias sret %result, i64 %top_left_x, i64 %top_left_y, i64 %len) {
    %square = alloca %Rect
    ; %square.tl = top_left
    %square_tl_x = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 0
    %square_tl_y = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 1
    store i64 %top_left_x, i64* %square_tl_x
    store i64 %top_left_y, i64* %square_tl_y

    ; %square.br.x = top_left + len
    %square_br_x = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 0
    %t1 = add i64 %top_left_x, %len
    store i64 %t1, i64* %square_br_x

    ; %square.br.y = top_left - len
    %square_br_y = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 1
    %t2 = sub i64 %top_left_y, %len
    store i64 %t2, i64* %square_br_y

    ; return square
    %result_tl_x = getelementptr %Rect, %Rect* %result, i32 0, i32 0, i32 0
    %result_tl_y = getelementptr %Rect, %Rect* %result, i32 0, i32 0, i32 1 ...
    %t3 = load i64, i64* %square_tl_x
    %t4 = load i64, i64* %square_tl_y ...
    store i64 %t3, i64* %result_tl_x
    store i64 %t4, i64* %result_tl_y ...
    ret void
}
```

getelementpointer

- The getelementpointer instruction handles indexing into tuple, array, and pointer types
 - Given a type, a pointer p of that type, and a *path* q consisting of a sequence of indices, getelementpointer computes the address of $p \rightarrow q$

- Does *not* access memory (like x86 lea)

```
%Point = type { i64, i64 }
```

```
%Rect = type { %Point, %Point }
```

getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types
 - Given a type, a pointer p of that type, and a *path* q consisting of a sequence of indices, `getelementpointer` computes the address of $p \rightarrow q$

- Does **not** access memory (like x86 `leaq`)

```
%Point = type { i64, i64 }
```

```
%Rect = type { %Point, %Point }
```

```
%square_tl_x = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 0  
               &(%square[0])  
               &(%square[0].tl)  
               &(%square[0].tl.x)
```

computes `%square + 0*sizeof(struct Rect) + 0 + 0`

getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types
 - Given a type, a pointer p of that type, and a *path* q consisting of a sequence of indices, `getelementpointer` computes the address of $p \rightarrow q$

- Does **not** access memory (like x86 `leaq`)

```
%Point = type { i64, i64 }
```

```
%Rect = type { %Point, %Point }
```

```
%square_tl_y = getelementptr %Rect, %Rect* %square, i32 0, i32 0, i32 1  
               &(%square[0])  
               &(%square[0].tl)  
               &(%square[0].tl.y)
```

computes `%square + 0*sizeof(struct Rect) + 0 + sizeof(i64)`

getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types
 - Given a type, a pointer p of that type, and a *path* q consisting of a sequence of indices, `getelementpointer` computes the address of $p \rightarrow q$

- Does **not** access memory (like x86 `leaq`)

```
%Point = type { i64, i64 }
```

```
%Rect = type { %Point, %Point }
```

```
%square_br_y = getelementptr %Rect, %Rect* %square, i32 0, i32 1, i32 1  
               &(%square[0])  
               &(%square[0].br)  
               &(%square[0].br.y)
```

computes `%square + 0*sizeof(struct Rect) + sizeof(struct Point) + sizeof(i64)`

getelementpointer

- The `getelementpointer` instruction handles indexing into tuple, array, and pointer types
 - Given a type, a pointer p of that type, and a *path* q consisting of a sequence of indices, `getelementpointer` computes the address of $p \rightarrow q$

- Does **not** access memory (like x86 `leaq`)

```
%Point = type { i64, i64 }
```

```
%Rect = type { %Point, %Point }
```

```
%square6_br_y = getelementptr %Rect, %Rect* %square, i32 6, i32 1, i32 1  
                  &(%square[6])  
                  &(%square[6].tl)  
                  &(%square[6].tl.y)
```

computes `%square + 6*sizeof(struct Rect) + sizeof(struct Point) + sizeof(i64)`

Arrays

Single-dimensional arrays

- In C: essentially the same as tuples
 - Array is stored as a contiguous chunk of memory
 - Index into position of i of an array a of t s with $a + \text{sizeof}(t)*i$

Single-dimensional arrays

- In C: essentially the same as tuples
 - Array is stored as a contiguous chunk of memory
 - Index into position of i of an array a of t s with $a + \text{sizeof}(t)*i$
- Memory-safe languages (e.g. OCaml & Java) must check that an array access is within bounds before accessing
 - Compiler must generate array access checking code
 - Store array length before array contents, or in a pair
 - type bytes = char array $\rightarrow$ %bytes = type { i64, [0 x i8] }*
 - or %bytes = type { i64, i8* }*

Single-dimensional arrays

- In C: essentially the same as tuples
 - Array is stored as a contiguous chunk of memory
 - Index into position of i of an array a of t s with $a + \text{sizeof}(t)*i$
- Memory-safe languages (e.g. OCaml & Java) must check that an array access is within bounds before accessing
 - Compiler must generate array access checking code
 - Store array length before array contents, or in a pair
 - type `bytes = char array` $\rightarrow$ `%bytes = type { i64, [0 x i8] }*`
 - or `%bytes = type { i64, i8* }*`
 - Example: suppose we want to load `a[i]` into `%rax`; suppose `%rbx` holds a pointer to `a` and `%rcx` holds an index.

```
movq (%rbx), %rdx      // load size into rdx
cmpq %rdx, %rcx        // compare index to bound
j l __ok               // jump if i < a.size
callq __err_oob         // test failed, call the error handler
__ok:
movq 8(%rbx, %rcx, 8) %rax // load a[i]
```

Multi-dimensional arrays

- In C: row-major order
 - 3x2 array: `m[0][0]`, `m[0][1]`, `m[1][0]`, `m[1][1]`, `m[2][0]`, `m[2][1]`
- In Fortran: column-major order
 - 3x2 array: `m[0][0]`, `m[1][0]`, `m[2][0]`, `m[0][1]`, `m[1][1]`, `m[2][1]`
- In OCaml & Java: no multi-dimensional arrays
 - 2-dimensional array is an array of arrays
`type mat = int array array` $\rightarrow$ `%mat = type { i64, { i64, i64* }* }`

Strings

- Null-terminated arrays of characters
- String constants are usually kept in **read only** segment (immutable!)
 - LLVM: `@str = constant [18 x i8] c"Factorial is %ld\0A\00"`
 - X86: `str: .string "Factorial is %d\n"`

Variant types

Enumerations

- `type color = Red | Green | Blue → i8`
 - Red → 0
 - Green → 1
 - Blue → 2

Enumerations

- type color = Red | Green | Blue $\rightarrow$ i8
 - Red $\rightarrow$ 0
 - Green $\rightarrow$ 1
 - Blue $\rightarrow$ 2
- Compiling switch:
 - 1 Nested if statements
 - 2 Jump tables (for dense switches):

```
switch(color) {  
  case Red:  
    ...  
  case Green:  
    ...  
  case Blue:  
    ...  
}
```



```
#color in %rax  
jmp (table, %rax, 8)  
LabelRed:  
...  
LabelGreen:  
...  
LabelBlue:  
...  
table:  
    .quad LabelRed, LabelGreen, LabelBlue
```

Algebraic data types

- Algebraic data types hold data, and can pattern match on constructor
- **type** *expr* = *Add* of *expr* * *expr* | *Var* of *string*
 - Easy way: quadword tag + payload. Must store a pointer if more space is needed.
 - *type* %*expr* = { **i64**, **i64*** }
 - (use bitcast to convert **i64*** pointer to { %*expr**, %*expr** }* or { **i64**, [0 x **i8**] }* after pattern matching)
 - More complicated way: tack a quadword tag in front of payload

Algebraic data types

- Algebraic data types hold data, and can pattern match on constructor
- **type** *expr* = *Add* of *expr* * *expr* | *Var* of *string*
 - Easy way: quadword tag + payload. Must store a pointer if more space is needed.
 - *type* %*expr* = { **i64**, **i64*** }
 - (use bitcast to convert **i64*** pointer to { %*expr**, %*expr** }* or { **i64**, [0 x **i8**] }* after pattern matching)
 - More complicated way: tack a quadword tag in front of payload
- Nested pattern matching → unnested pattern matching at AST level

Compiler phases (simplified)

