SWEN430 - Compiler Engineering

Lecture 17 - Machine Code III

Lindsay Groves and David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Switch Statements

For performance, `switch` implemented with **jump tables**:

```
int main(int argc, char** argv) {
  switch(argc - 1) {
     case 1:
        return 0;
     case 2:
        return 1;
     default:
        return -1;
} }
```

Switch Statements

For performance, `switch` implemented with **jump tables**:

```
    .text
    .globl main
main:
    movq %rdi, %rbx // read argc into rbx
    dec %rbx        // argc - 1
    cmpq 0, %rbx
    jle CD          // if <=0
    cmpq 2, %rbx
    jg CD           // if > 2
    movq JUMPTABLE(,%rbx,8), %rbx
    jmp *%rbx       // jump to place
JUMPTABLE:
    .quad CD        // pad 0 value
    .quad L1        // offset 1
    .quad L2        // offset 2
CD:
    movq -1, %rax  // -1 into return register
    jmp END
L1:
    movq 0, %rax   // 0 into return register
    jmp END
L2:
    movq 1, %rax   // 1 into return register
    jmp END
END:
    ret             // return / end
```

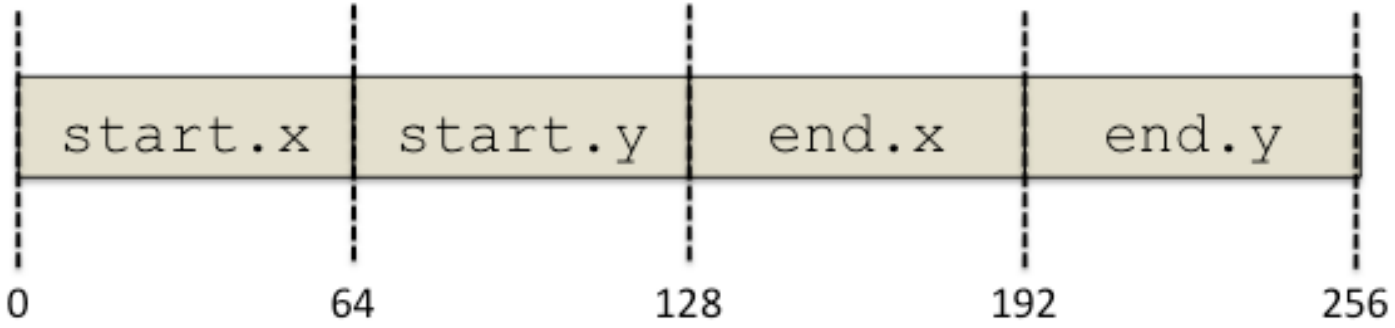
Data Representation — Overview

- Need to represent WHILE **data types** in memory!
- Some data types have **fixed widths**
  - » e.g. `int` is 64 bits wide (on x86_64)
  - » e.g. `char` is 8 bits wide (ASCII)
  - » e.g. `{int f, int g}` is 128 bits wide (on x86_64)
- Other data types have **variable widths**
  - » e.g. `int[]` has variable width
  - » e.g. `{int[] array}` has variable width
- In WHILE, arrays are **only source** of variable-width types

Data Representation — Records

- Ignoring arrays, records have **statically determinable** widths
- Records in WHILE can be **flattened** into a contiguous sequence:

```
type Point is {int x, int y}
type Line is {Point start, Point end}
```

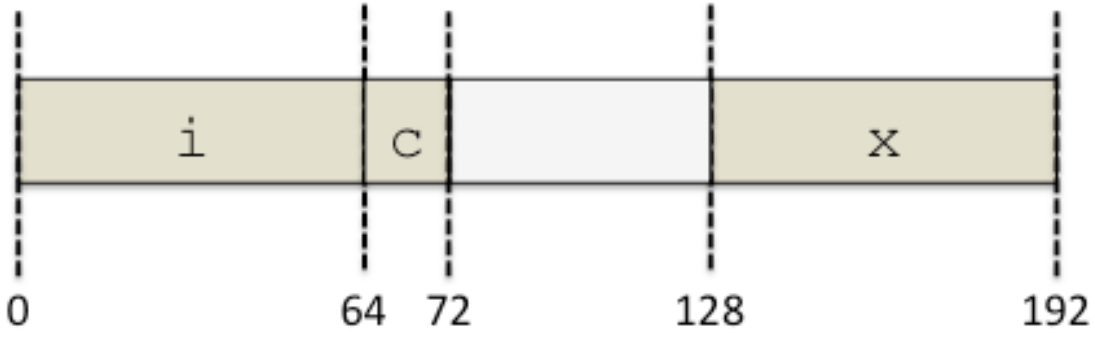


- Every field has **fixed offset** so can exploit addressing modes
  - » e.g. `v = l.end.x` might translate to `mov 16(%rdi),%eax`
- Fields sorted **alphabetically** so declaration order independent

Data Representation — Alignment

- Can using padding to ensure fields are **aligned**

```
type Line is {int i, char c, int x}
```



- Alignment is **unnecessary** but can **improve performance**
  - » Because aligned memory accesses are typically faster

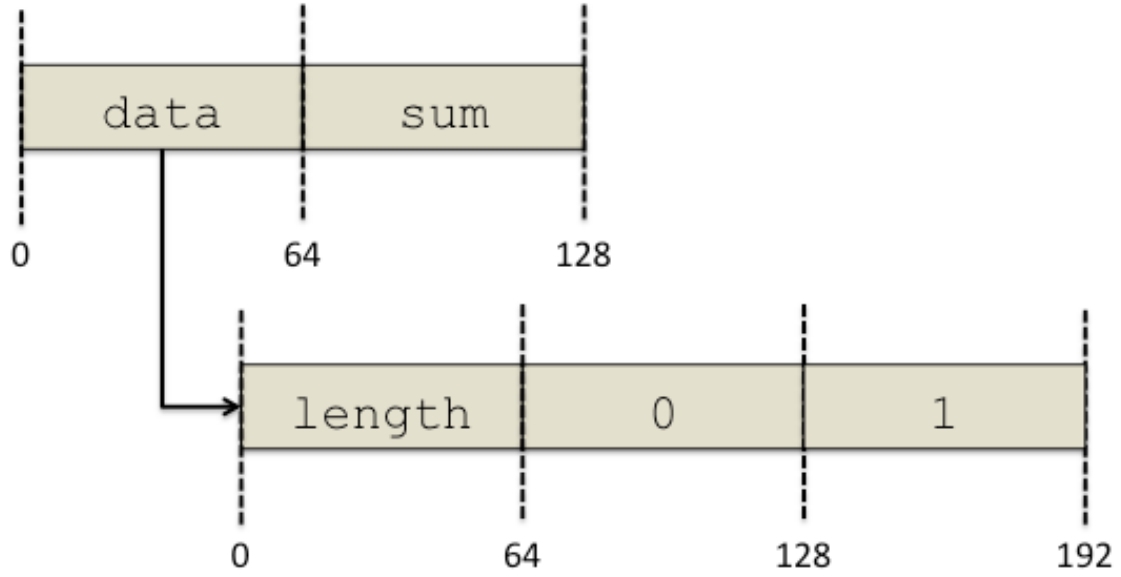
- But, alignment can also **reduce performance**!
  - » e.g. if record no longer fits in a single **cache line**

Data Representation — Variable Width

- For arrays, we **cannot predetermine** their width:

```
type SumList is {int[] data, int sum}
```

- Instead, arrays are **dynamically allocated**:



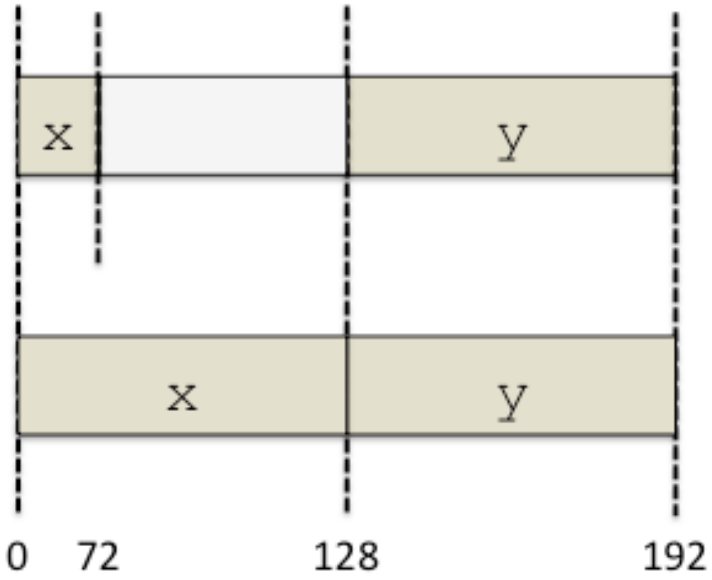
- Enclosing record has fixed width with array represented as **pointer**

- This makes dealing with **value semantics** quite tricky!

Data Representation — Untagged Unions

- For union can compute **maximum** size of elements

```
type SumList is {int x,int y} | {char x, int y}
```



- This is how **unions** of **structs** are implemented in C

- Common elements accessible via **common initial sequence**

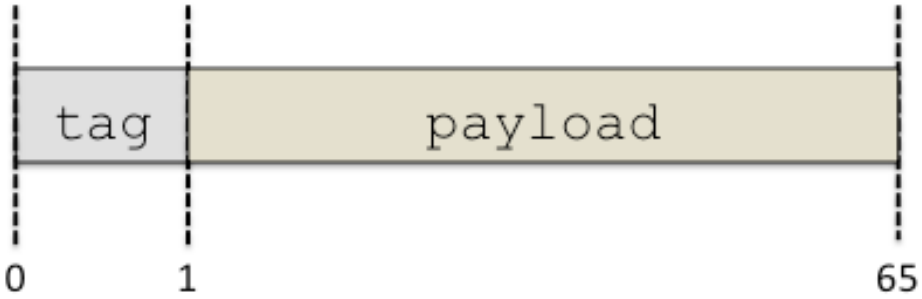
Data Representation — Tagged Unions

- Need a way to **determine at runtime** what a union is:

```
int f(int|null item) {
    if(item is int) { return (int) item; }
    else { return 0; }
}
```

- If `sizeof(item) == sizeof(int)` then **cannot** to do this!

- Instead, need to add special **tag** field to representation of `item`:



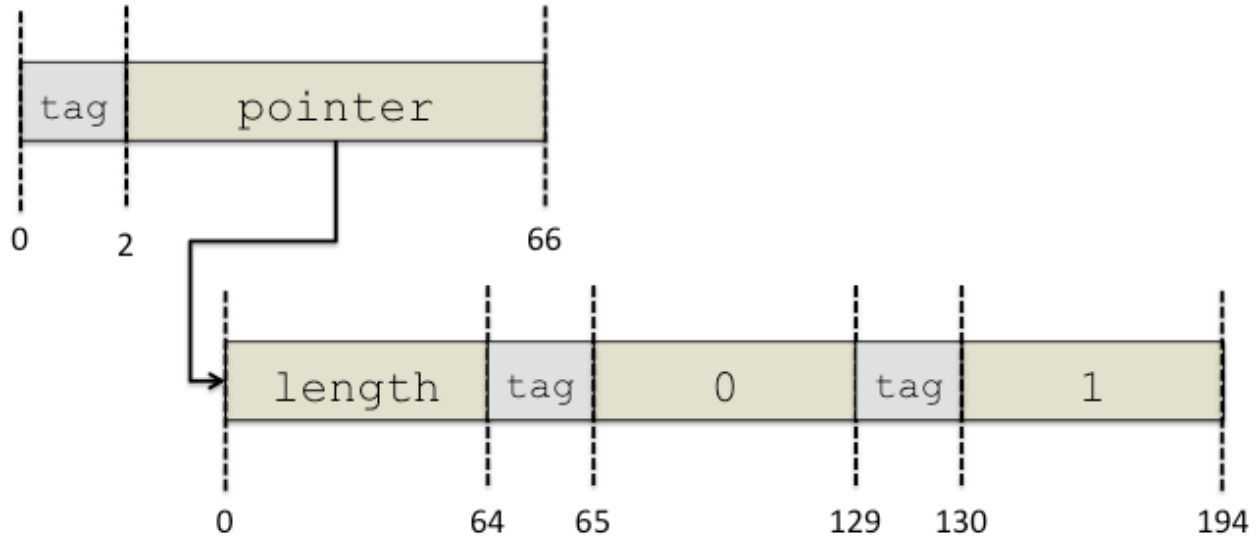
- Here, only **1 bit tag** required; in general, depends on number of **distinct types**

Data Representation — Tagged Unions

```
type NullPoint as {int|null x, int|null y}
```

- How many tags bits **required** to represent a `NullPoint`?
- Representing arrays is slightly **more complicated**:

```
type NullList as (int|null)[]
```



- Why might we choose to add **two tag bits** for the pointer as well?

Data Representation — Retagging

- Consider this example:

```
int|null f(int x) {
  return x;
}
```

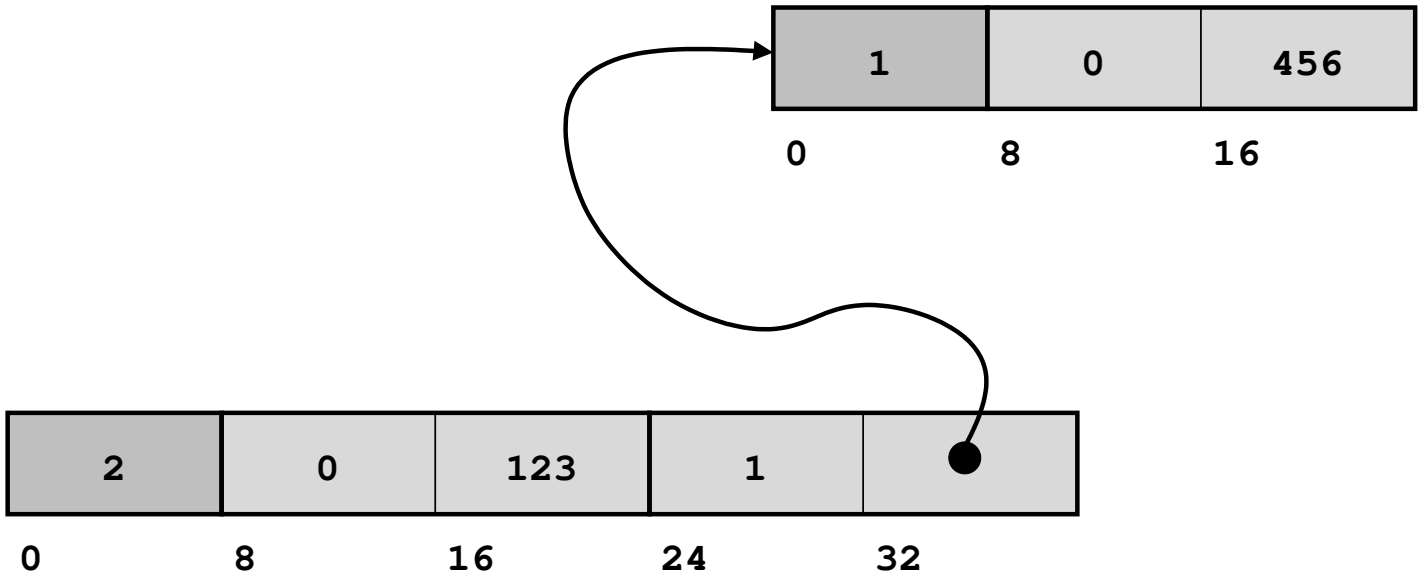
- In the above, need to **initialise** tag for return value

- Now, consider this example:

```
int|null|char f(int|null x) {
  return x;
}
```

- In this case, we need to **retag** variable `x` on return

Data Representation — Uniform



- WHILE uses **uniform** representation for records and arrays
- This is **inefficient** — *Why?*
- But, **simplifies** common operations (e.g. *equality*, *cloning*)

CDecl Calling Convention

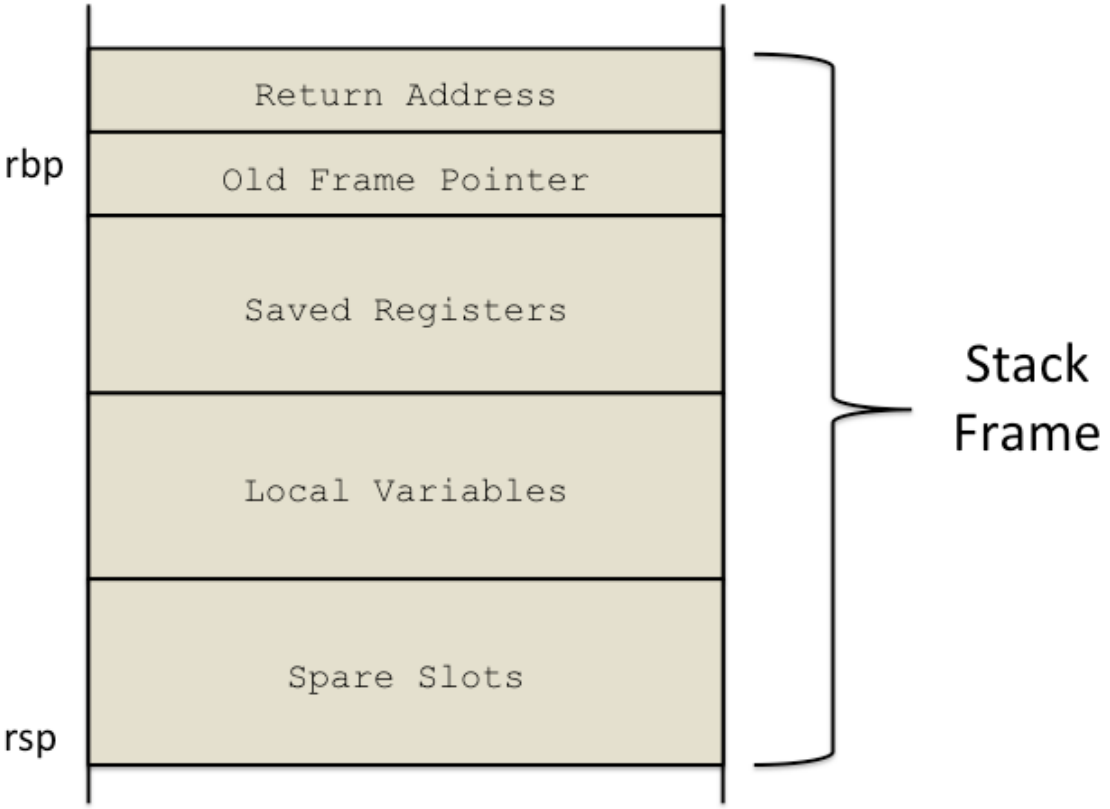
- The `cdecl` calling-convention commonly used for **C programs**
- Used for C because supports **variable length** arguments
- In this convention:
  - » **Caller responsible** for cleaning parameters off stack
  - » Registers `rax`, `rcx` and `rdx` are **caller saved**, all others **callee saved**
  - » **Arguments** passed on the stack in right-to-left order
  - » **Return value** passed back in `rax`

Register Spilling

- Given **limited registers** on x86, we'll eventually run out of them!
- When this happens, can fall back to using **stack-based approach**
- More efficient to calculate number of **additional "registers" required**
- Then, **pre-allocate** stack space for additional "overflow" registers

Register Spilling (Continued)

- This is roughly what the **stack frame** will look like then:



- Note: not always possible to **predetermine slots**

Omitting the Frame Pointer

- GCC optimisation is **omitting** frame pointer when possible.
- This can cause problems with **debuggers**.
- For 32 bit machines, **only** 8 registers.
- Having frame pointer loses a register (and `esp` already taken).
- Minor problem with 64 bit machines as have **more registers**.