CS443: Compiler Construction

Lecture 7: Advanced control flow

Stefan Muller

While loops have a backward jump

```
while (x < 10) s1

testl:
  %temp = icmp lt i32 %x 10
  br i1 %temp, label %bodyl, label %donel
bodyl:
  (compilation of s1)
  br label %testl
donel:
```

Unconditional jump back to test (NOT start of body!)

For break and continue, keep track of the “test” and “done” labels

```
while (x < 10) if (x < 5) break;

testl:  %temp = icmp lt i32 %x 10
        br i1 %temp, label %bodyl, label %donel
bodyl:  %temp2 = icmp lt i32 %x 5
        br i1 %temp2, label %truel, label %falsel
truel:  br label %donel    <- Break: jump to done
        br label %endif    <- Problem: Not a valid basic block
falsel: br label %endif
endif:  br label %testl
donel: …
```

Hacky solution: Put a dummy label after break/continue/return

```
while (x < 10) if (x < 5) break;

testl:  %temp = icmp lt i32 %x 10
        br i1 %temp, label %bodyl, label %donel
bodyl:  %temp2 = icmp lt i32 %x 10
        br i1 %temp, label %truel, label %falsel
truel:  br label %donel
lbl123: br label %endif
falsel: br label %endif
endif:  br label %testl
donel: …
```

Unreachable basic block

For continue, jump to test

```
while (x < 10) if (x < 5) continue;

testl:  %temp = icmp lt i32 %x 10
        br i1 %temp, label %bodyl, label %donel
bodyl:  %temp2 = icmp lt i32 %x 10
        br i1 %temp, label %truel, label %falsel
truel:  br label %testl
lbl123: br label %endif
falsel: br label %endif
endif:  br label %testl
donel: …
```

`Call` calls a function

%dest = call <retty> <funptr>(<ty1> <arg1>, …, <tyN> <argN>)

e.g.

- %res = call i32 @abs(i32 -5)
- %ptr = call i8* @malloc(i32 256)

`Call` calls a function

- Does a lot!
  - Push new stack frame
  - Copy over args
  - (Save registers)
  - Jump
- Not a terminator—it returns a value!

`Call` calls a function

%dest = call <retty> <funptr>(<ty1> <arg1>, …, <tyN> <argN>)

<funptr> is a **variable** (not constant!) with the address of a function

e.g., `@malloc` is a global var with the address of `malloc`

```
define i32 @call42(i32(i32)* %f) {
  %temp = call i32 %f(i32 42)
  ret i32 %temp
}
```

`Switch` is relatively easy (in LLVM)

```
switch x {
  case 0:
    s0
  case 1:
    s1
…
  default:
    sd
}
```

```
  switch i32 %x, label %ldefault
    [ i32 0, label %l0
      i32 1, label %l1
      …
    ]
l0:
 (Compilation of s0)
  br %l1          ← Implement fall-through
l1: …
ldefault:
 (Compilation of sd)
  br %ldone
ldone:
```

`Switch` is relatively easy (in LLVM)

```
switch x {
  case 0:
    s0
    break;
  case 1:
    s1
…
  default:
    sd
}
```

```
  switch i32 %x, label %ldefault
    [ i32 0, label %l0
      i32 1, label %l1
      …
    ]
l0:
 (Compilation of s0)
  br %ldone
lbl123: br %l1
l1: …
ldefault:
 (Compilation of sd)
  br %ldone
ldone:
```

What if we didn’t have LLVM **`switch`**? Option #1: Convert to if

```
switch x {
  case 0:
    s0
  case 1:
    s1
…
  default:
    sd
}
```

```
if (x == 0) s0
else if (x == 1) s1
…
else sd
```

More efficient if many cases: Binary search

What if we didn’t have LLVM `switch`? Option #2: Jump table (array of labels)

```
switch x {
  case 0:
    s0
  case 1:
    s1
…
  default:
    sd
}
```

```
if 0 <= x <= 1: Branch to label at jt[x]
else: Branch to %ldefault
l0:
 (Compilation of s0)
   br %l1
l1: …
ldefault:
 (Compilation of sd)
   br %ldone
ldone:
```

jt	0	1
	%l0	%l1

What if we didn't have LLVM **`switch`**? Option #2: Jump table (array of labels)

```
switch x {
  case 0:
    s0
  case 1000:
    s1
…
  default:
    sd
}
```

```
if 0 <= x <= 1000: Branch to label at jt[x]
else: Branch to %ldefault
l0:
 (Compilation of s0)
   br %l1
l1: …
ldefault:
 (Compilation of sd)
   br %ldone
ldone:
```

Option #1 probably better

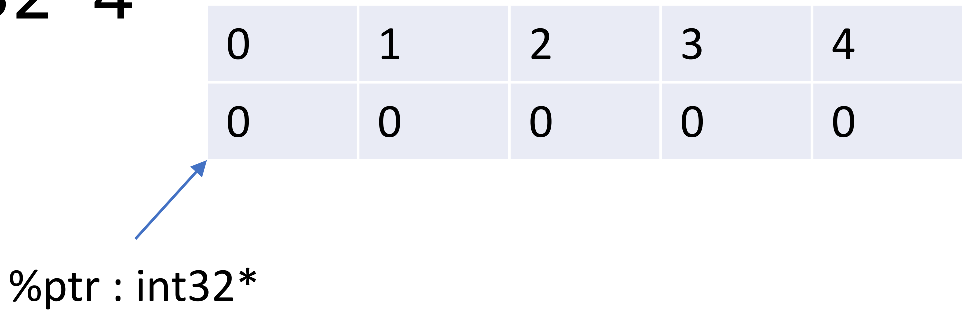
jt

0	1	2	…	999	1000
%l0	%l1	%ldef		%ldef	%l1

Arrays in LLVM

- LLVM has built-in arrays.
- We’re not going to use them.
- Instead: pointers, like in C

```
%ptr = alloca int32, int32 4
```



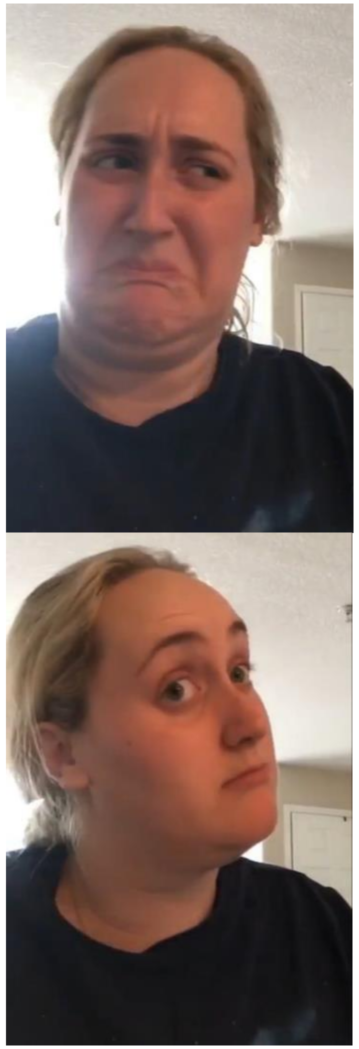
Get/set array elements with `load/store`

```
int[4] ptr;
ptr[0] = 42;
a = ptr[0];
```

```
%ptr = alloca i32, i32 4
store i32 42, i32* %ptr
%a = load i32, i32* %ptr
```

How to get the address of ptr[x]?

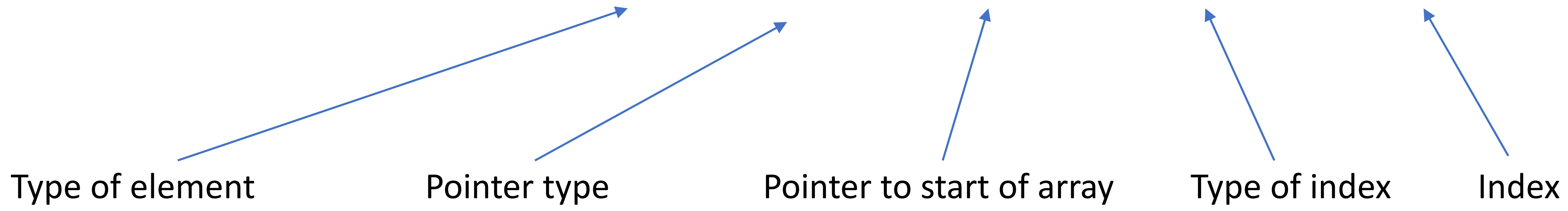
- Pointer arithmetic?



- `getelementptr`

`getelementptr` (for arrays)

```
%elptr = getelementptr <ty>, <ty>* %ptr, <intty> <val>
```



Ex.

```
%el5 = getelementptr i32, i32* %ptr, i32 5
```

(address of %ptr[5])

Returns **address** of element. Doesn’t do load/store

Get/set array elements with `load/store`

```
int[4] ptr;
ptr[x] = 42;
a = ptr[y];
```

```
%ptr = alloca i32, i32 4
%elx = getelementptr i32, i32* %ptr, i32 %x
store i32 42, i32* %elx
%ely = getelementptr i32, i32* %ptr, i32 %y
%a = load i32, i32* %ptr
```