Lecture 17: Pointers, arrays, and structs

David Hovemeyer

October 27, 2025

601.428/628 Compilers and Interpreters

Today

- Functions and function calls
- Pointers and lvalues
- Arrays
- Structs

Functions and function calls

Function calls

Generating code for functions and function calls is reasonably straightforward.

Main issue: the argument registers need to be available to use to pass argument values to the called function.

- And, a function containing function calls will need to retrieve its own arguments from the argument registers

Note that on some architectures (e.g., 32-bit x86), arguments are passed on the stack (and are accessed in memory via the stack pointer.)

Also, even on architectures where argument registers are used, there will be a fixed number of them, and "excess" arguments will typically be passed on the stack.

Argument registers

In the high-level IR, virtual registers `vr1` through `vr9` are used to pass argument values to functions.

In the low-level (x86-64) IR, the argument registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass argument values to functions. (You won't be required to support functions with more than 6 parameters.)

Storage for function parameters

Function parameters are variables which should be allocated storage in the same way as any other local variable.

At the beginning of the code generated for a function, the high-level code generator should emit a series of `mov` instructions to copy the value of each argument register into its corresponding parameter variable.

This ensures that the parameter values can be accessed in the body of the function, even though the argument registers might need to be used to pass arguments to called functions.

Moving argument values to parameter variables (example)

```
/* C code */
int sum_arr(int *arr, int n) {
  int i, sum;
  sum = 0;
  for (i = 0; i < n; i = i + 1)
    sum = sum + arr[i];
  return sum;
}
```

```
/* high-level IR */
sum_arr:
    enter     $0
    mov_q     vr10, vr1
    mov_l     vr11, vr2
    mov_l     vr14, $0
    mov_l     vr13, vr14
    mov_l     vr15, $0
    mov_l     vr12, vr15
    ...etc...
```

Note: `arr` is `vr10`, `n` is `vr11`

Calling a function

Calling a function means

1. Evaluating argument expressions and moving their values to the argument registers (`vr1`, `vr2`, etc.)
2. Emitting a `call` instruction

Calling a function (example)

```
/* C code */
int add(int a, int b);

int main(void) {
  int sum;
  sum = add(3, 4);
  return sum;
}
```

```
/* high-level IR */
main:
     enter     $0
     mov_l     vr11, $3
     mov_l     vr1, vr11
     mov_l     vr12, $4
     mov_l     vr2, vr12
     call      add
     ...etc...
```

Returning a value

In the high-level IR, the computed return value should be assigned to `vr0` (the return value register).

Since a C function could have multiple `return` statements, including "early" returns, this should be followed by jumping to a label marking the function epilogue and `ret` instruction.

Returning a value (example)

```
/* C code */
int add(int a, int b);

int main(void) {
  int sum;
  sum = add(3, 4);
  return sum;
}
```

```
/* high-level IR */
main:
    enter   $0
    ...compute arg values...
    call    add
    mov_l   vr13, vr0
    mov_l   vr10, vr13
    mov_l   vr0, vr10
    jmp     .Lmain_return
.Lmain_return:
    leave   $0
    ret
```

Capture return value of `add` function, store it in `vr0` to return from `main`

Returning a value (example)

```
/* C code */
int add(int a, int b);

int main(void) {
  int sum;
  sum = add(3, 4);
  return sum;
}
```

Jump to function epilogue

```
/* high-level IR */
main:
    enter    $0
    ...compute arg values...
    call     add
    mov_l    vr13, vr0
    mov_l    vr10, vr13
    mov_l    vr0, vr10
    jmp      .Lmain_return
.Lmain_return:
    leave    $0
    ret
```

Function calls in low-level code

Because all variables have storage locations independent from argument registers, the low-level argument registers (`%rdi`, `%rsi`, etc.) are always available for use by the low-level code generator.[1]

The return value register `%rax` is also always available.

So:

- The high-level argument registers (`vr1` through `vr6`) are synonymous with the low-level argument registers (`%rdi`, `%rsi`, etc.)
- The high-level return value register `vr0` is synonymous with the lowel-level return value register `%rax`

[1] The register allocator will also be involved in their use, more about that soon.

Function call in low-level code (example)

```
/* C code */
int add(int a, int b);

int main(void) {
  int sum;
  sum = add(3, 4);
  return sum;
}
```

Compute arguments, place in argument registers, call function, store result in sum

```
/* low-level IR */
main:
  pushq %rbp                /* enter $0 */
  movq  %rsp, %rbp
  subq  $32, %rsp
  movl  $3, -24(%rbp)       /* mov_l vr11, $3 */
  movl  -24(%rbp), %edi     /* mov_l vr1, vr11 */
  movl  $4, -16(%rbp)       /* mov_l vr12, $4 */
  movl  -16(%rbp), %esi     /* mov_l vr2, vr12 */
  call  add                 /* call  add */
  movl  %eax, -8(%rbp)      /* mov_l vr13, vr0 */
  movl  -8(%rbp), %r10d     /* mov_l vr10, vr13 */
  movl  %r10d, -32(%rbp)
  movl  -32(%rbp), %eax     /* mov_l vr0, vr10 */
  jmp   .Lmain_return       /* jmp   .Lmain_return */
.Lmain_return:
  addq  $32, %rsp           /* leave $0 */
  popq  %rbp
  ret                       /* ret    */
```

Function return in low-level code (example)

```
/* C code */
int add(int a, int b);

int main(void) {
  int sum;
  sum = add(3, 4);
  return sum;
}
```

Move sum to return value register, jump to function epilogue

```
/* low-level IR */
main:
  pushq %rbp               /* enter $0 */
  movq  %rsp, %rbp
  subq  $32, %rsp
  movl  $3, -24(%rbp)      /* mov_l vr11, $3 */
  movl  -24(%rbp), %edi    /* mov_l vr1, vr11 */
  movl  $4, -16(%rbp)      /* mov_l vr12, $4 */
  movl  -16(%rbp), %esi    /* mov_l vr2, vr12 */
  call  add                /* call  add */
  movl  %eax, -8(%rbp)     /* mov_l vr13, vr0 */
  movl  -8(%rbp), %r10d    /* mov_l vr10, vr13 */
  movl  %r10d, -32(%rbp)
  movl  -32(%rbp), %eax    /* mov_l vr0, vr10 */
  jmp   .Lmain_return      /* jmp   .Lmain_return */
.Lmain_return:
  addq  $32, %rsp          /* leave $0 */
  popq  %rbp
  ret                      /* ret     */
```

Function calls and register allocation

Looking ahead: *register allocation* is an optimization technique in which values computed and used in a function are stored in CPU registers.

The more CPU registers available to the register allocator, the more effective it can be.

For temporary values (e.g., values of evaluated subexpressions), caller-saved registers are usually appropriate.

It is beneficial to make argument registers (which are caller-saved) available to the register allocator. However, this needs to be done in a way that doesn't interfere with their use to pass arguments to called functions.

Local register allocation and function calls

In *local register allocation*, register allocation decisions are made at the level of individual basic blocks. This is relatively straightforward to do. (Contrast: *global register allocation*, which can be more effective overall, but which requires more powerful analysis to do safely.)

Idea: the local register allocator can use the argument registers to store temporary values as long as they won't be needed for a function call (if the basic block contains one.)

More about this later when we cover register allocation.

Pointers and lvalues

Pointers and lvalues

A *pointer* value represents the memory address of an lvalue's storage location.

A variable is an lvalue. So are fields of structs and array elements.

For variables requiring storage in memory, the storage allocator should assign it a storage offset in the stack frame. General requirements:

- variables with overlapping lifetimes require non-overlapping storage
- storage offsets must reflect alignment requirements of the variable's data type

Codegen for lvalues

Important idea: if the generated code needs to refer to an lvalue, the code generator must be able to place the address of its storage location in a register. (For the high-level code generator, it must place the lvalue's storage address in a virtual register.)

So, lvalues are ultimately about *computing* the address a storage location:

- Easy case: for a variable reference, the code generator knows its offset in the local storage area.[2] Emit a `localaddr` instruction to load the lvalue's storage address into a temporary virtual register.
- Array subscript: compute the address of an array element by adding a computed offset to a base address.
- Field reference: compute the address of a struct member by adding its offset to a base address pointing to the beginning of the struct.

[2] The offset should be in the variable's symbol table entry.

Representing lvalues during high-level codegen

Recall that the high-level code generator, when generating code for an expression, will put an `Operand` in the AST node representing the expression to indicate how to access the result of the expression.

- E.g., if `vr15` is where the result of the expression was placed, then the `Operand` representing `vr15` is placed in the AST node

Idea: the `Operand` for an expression yielding an lvalue is a memory reference operand of the form (vr$N$), where N is the register number of the virtual register containing the address of the lvalue's storage location.

- For "scalar" (integral value or pointer) lvalues, the operand *is* the correct way to refer to the lvalue
- For arrays and struct instances, the operand can be "unpacked" to get the virtual register containing the base address

Address-of, pointer dereference

Using the representation `(vr`N`)` for lvalues has a nice benefit:

- The C address-of operator (`&`) means transforming `(vr`N`)` into `vr`N
- The C pointer dereference operator (`*`) means transforming `vr`N into `(vr`N`)`

Pointer operations (example)

```
/* C code */
int read_i32(void);
void print_i32(int x);

int main(void) {
  int a, b, *p;
  a = read_i32();
  b = read_i32();

  if (a < b)
    p = &a;
  else
    p = &b;
  *p = 42;

  print_i32(a);
  print_i32(b);
  return 0;
}
```

```
/* high-level IR */
```

Note: `p` is `vr10`, a's storage is at offset 0, b's storage is at offset 4

Pointer operations (example)

```
/* C code */
int read_i32(void);
void print_i32(int x);

int main(void) {
  int a, b, *p;
  a = read_i32();
  b = read_i32();

  if (a < b)
    p = &a;
  else
    p = &b;
  *p = 42;

  print_i32(a);
  print_i32(b);
  return 0;
}
```

```
/* high-level IR */
call      read_i32
mov_l     vr11, vr0
localaddr vr12, $0
mov_l     (vr12), vr11
```

Store result of calling `read_i32()` in `a`

Note: `p` is `vr10`, `a`'s storage is at offset 0, `b`'s storage is at offset 4

Pointer operations (example)

```
/* C code */
int read_i32(void);
void print_i32(int x);

int main(void) {
  int a, b, *p;
  a = read_i32();
  b = read_i32();

  if (a < b)
    p = &a;
  else
    p = &b;
  *p = 42;

  print_i32(a);
  print_i32(b);
  return 0;
}
```

```
/* high-level IR */
localaddr vr15, $0
localaddr vr16, $4
mov_l     vr18, (vr15)
mov_l     vr19, (vr16)
cmplt_l   vr17, vr18, vr19
```

Determine whether the value in a is less than the value in b

Note: p is vr10, a's storage is at offset 0, b's storage is at offset 4

Pointer operations (example)

```
/* C code */
int read_i32(void);
void print_i32(int x);

int main(void) {
  int a, b, *p;
  a = read_i32();
  b = read_i32();

  if (a < b)
    p = &a;
  else
    p = &b;
  *p = 42;

  print_i32(a);
  print_i32(b);
  return 0;
}
```

```
/* high-level IR */
localaddr vr20, $0
mov_q     vr10, vr20
```

Store the address of a in p

Note: `p` is `vr10`, a's storage is at offset 0, b's storage is at offset 4

Pointer operations (example)

```
/* C code */
int read_i32(void);
void print_i32(int x);

int main(void) {
  int a, b, *p;
  a = read_i32();
  b = read_i32();

  if (a < b)
    p = &a;
  else
    p = &b;
  *p = 42;

  print_i32(a);
  print_i32(b);
  return 0;
}
```

```
/* high-level IR */
mov_l    vr22, $42
mov_l    (vr10), vr22
```

Store the value 42 in the variable that p points to

Note: p is vr10, a's storage is at offset 0, b's storage is at offset 4

Arrays

Arrays

C defines the array subscript operator `a[i]` as being equivalent to `*(a + i)`.

So, accessing an array element is based on *pointer arithmetic*.

When used in an expression, an array "decays" to a pointer to its first element.

If `a` points to the first element of an array, `a + i` computes a pointer to the element at index `i` of that array.

- I.e., `a + i` is `&a[i]`

Pointer arithmetic

To evaluate the result of a pointer arithmetic expression `a + i`, the generated code should

- Compute the *offset* of the element `i` positions from the one `a` points to; this is `i` multiplied by the element size (i.e., the size of the data type that the pointer `a` points to)
- Add the offset to `a`; the sum is the result

Pointer arithmetic (example)

```
/* C code */
int read_i32(void);

int main(void) {
  int arr[10], i, *p;
  i = read_i32();
  p = arr + i;
  *p = 42;
  return *p;
}
```

Note: `arr` is at offset 0, `i` is `vr10`, `p` is `vr11`

```
/* high-level IR */
main:
    enter     $40
    call      read_i32
    mov_l     vr12, vr0
    mov_l     vr10, vr12
    localaddr vr13, $0
    sconv_lq vr15, vr10
    mul_q     vr16, vr15, $4
    add_q     vr14, vr13, vr16
    mov_q     vr11, vr14
    mov_l     vr17, $42
    mov_l     (vr11), vr17
    mov_l     vr0, (vr11)
    jmp       .Lmain_return
.Lmain_return:
    leave     $40
    ret
```

Pointer arithmetic (example)

```
/* C code */
int read_i32(void);

int main(void) {
  int arr[10], i, *p;
  i = read_i32();
  p = arr + i;
  *p = 42;
  return *p;
}
```

Note: `arr` is at offset 0, `i` is `vr10`, `p` is `vr11`

Compute offset of element at index `i`, add to base address, put element address in `vr14`

```
/* high-level IR */
main:
    enter     $40
    call      read_i32
    mov_l     vr12, vr0
    mov_l     vr10, vr12
    localaddr vr13, $0
    sconv_lq vr15, vr10
    mul_q     vr16, vr15, $4
    add_q     vr14, vr13, vr16
    mov_q     vr11, vr14
    mov_l     vr17, $42
    mov_l     (vr11), vr17
    mov_l     vr0, (vr11)
    jmp       .Lmain_return
.Lmain_return:
    leave     $40
    ret
```

Pointer arithmetic (example)

```
/* C code */
int read_i32(void);

int main(void) {
  int arr[10], i, *p;
  i = read_i32();
  p = arr + i;
  *p = 42;
  return *p;
}
```

Note: `arr` is at offset 0, `i` is `vr10`, `p` is `vr11`

Store computed element address in p

```
/* high-level IR */
main:
    enter     $40
    call      read_i32
    mov_l     vr12, vr0
    mov_l     vr10, vr12
    localaddr vr13, $0
    sconv_lq  vr15, vr10
    mul_q     vr16, vr15, $4
    add_q     vr14, vr13, vr16
    mov_q     vr11, vr14
    mov_l     vr17, $42
    mov_l     (vr11), vr17
    mov_l     vr0, (vr11)
    jmp       .Lmain_return
.Lmain_return:
    leave     $40
    ret
```

Pointer arithmetic (example)

```
/* C code */
int read_i32(void);

int main(void) {
  int arr[10], i, *p;
  i = read_i32();
  p = arr + i;
  *p = 42;
  return *p;
}
```

Note: `arr` is at offset 0, `i` is `vr10`, `p` is `vr11`

Store 42 in the array element p points to

```
/* high-level IR */
main:
    enter     $40
    call      read_i32
    mov_l     vr12, vr0
    mov_l     vr10, vr12
    localaddr vr13, $0
    sconv_lq  vr15, vr10
    mul_q     vr16, vr15, $4
    add_q     vr14, vr13, vr16
    mov_q     vr11, vr14
    mov_l     vr17, $42
    mov_l     (vr11), vr17
    mov_l     vr0, (vr11)
    jmp       .Lmain_return
.Lmain_return:
    leave     $40
    ret
```

Array subscript

Code generated for `a[i]` should be the same as the code generated for `*(a + i)`:

1. Pointer arithmetic to compute address of element
2. Dereference computed pointer to element

Multidimensional arrays

Consider a two-dimensional array: `int a[10][5];`

The type of this array is "array of 10 elements of type array of 5 elements of type `int`".

Your code generator should not need to implement a special case for multidimensional arrays! As long as in the computation of an element address the index is scaled by the correct element size, the address computation will be accurate. In this example, the element size of the array `a` is 20 (5 `int` elements of size 4.)

Efficiency of element references

Consider the following C code and generated low-level code for the access to the array element at index `i`:

```
/* C code */
int sum(int *arr, int n) {
  int i, sum;
  sum = 0;
  for (i = 0; i < n; i = i + 1) {
    sum = sum + arr[i];
  }
  return sum;
}
```

```
/* Low-level code (loop body) */
movl    %ebx, %r10d
movslq  %r10d, %r10
movq    %r10, %r9
movq    %r9, %r10
imulq   $4, %r10
movq    %r10, %r8
movq    %r13, %r10
addq    %r8, %r10
movq    %r10, %rcx
movl    (%rcx), %edx
movl    %r12d, %r10d
addl    %edx, %r10d
movl    %r10d, %esi
movl    %esi, %r12d
```

Note: the register allocator has allocated CPU registers for temp values, `%r12d` has been allocated for `sum`, and `%r13` has been allocated for `arr`.

Explicit address computations

Emitting explicit multiply and add instructions for array element address computations

- Requires multiple instructions per array element reference
- Requires CPU registers to hold temporary values

The CPU instruction set may have addressing modes that are useful for array element references. Can we take advantage of them?

Peephole optimization to utilize addressing modes

Generated low-level code for the access to the array element at index `i`, before and after peephole optimization:

```
/* Low-level code (before) */
movl    %ebx, %r10d
movslq  %r10d, %r10
movq    %r10, %r9
movq    %r9, %r10
imulq   $4, %r10
movq    %r10, %r8
movq    %r13, %r10
addq    %r8, %r10
movq    %r10, %rcx
movl    (%rcx), %edx
```

```
/* Low-level code (after) */
movslq    %ebx, %r9
movl      (%r13,%r9,4), %edx
```

Note: the register allocator has allocated CPU registers for temp values, `%r12d` has been allocated for `sum`, and `%r13` has been allocated for `arr`.

Idioms in generated code

Explicit address computation results in predictable *idioms* in the generated code.

Peephole optimization can recognize these idioms and replace them with more efficient ones.

Structs

Structs

For a struct type, the compiler needs to determine

- An offset for each field
- The number of bytes required to represent an instance of a struct

Field offsets must be guarantee alignment according to their data types.

- E.g., a field with type `int` field must have an offset that is a multiple of 4, since `sizeof(int)` $= 4$

Generally, a struct instance is similar to an array.

- It's an lvalue, and generated code will need to know which virtual register contains its base address
- Addresses of a field in a struct instance can be computed by adding its offset to the instance's base address

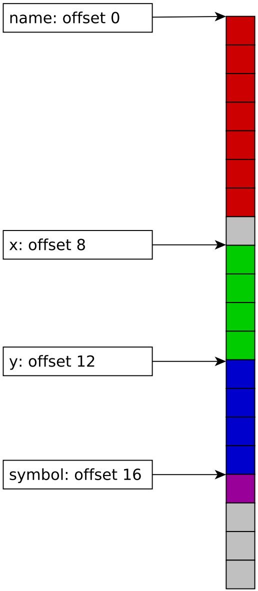
Example struct layout

```
struct Player {
   char name[7];
   int x, y;
   char symbol;
};
```

One padding byte is needed between name and x fields to ensure x has an offset that is a multiple of 4.

Three padding bytes are needed at end of struct (after the symbol field) to ensure that instances have addresses aligned on a multiple of 4.

An instance of `struct Player` requires 20 bytes of storage in memory aligned on an address that is a multiple of 4.



Struct type (example)

```
int read_i32(void);
void strcpy(char *dst,
            const char *src);

struct Player {
  char name[7];
  int x, y;
  char symbol;
};

int main(void) {
  struct Player p;
  strcpy(p.name, "Frodo");
  p.x = 17;
  p.y = 42;
  p.symbol = '@';
  return 0;
}
```

Struct type (example)

```
int read_i32(void);
void strcpy(char *dst,
            const char *src);

struct Player {
  char name[7];
  int x, y;
  char symbol;
};

int main(void) {
  struct Player p;
  strcpy(p.name, "Frodo");
  p.x = 17;
  p.y = 42;
  p.symbol = '@';
  return 0;
}
```

```
/* high-level IR */
localaddr vr10, $0
mov_q     vr11, $0
add_q     vr12, vr10, vr11
mov_q     vr1, vr12
mov_q     vr13, $_str0
mov_q     vr2, vr13
call      strcpy
```

Pass address of `p.name` as first argument to `strcpy` (p's storage is at offset 0 in the stack frame, `name` is at offset 0 in `p`)

Struct type (example)

```
int read_i32(void);
void strcpy(char *dst,
            const char *src);

struct Player {
  char name[7];
  int x, y;
  char symbol;
};

int main(void) {
  struct Player p;
  strcpy(p.name, "Frodo");
  p.x = 17;
  p.y = 42;
  p.symbol = '@';
  return 0;
}
```

```
/* high-level IR */
mov_l     vr14, $17
localaddr vr15, $0
mov_q     vr16, $8
add_q     vr17, vr15, vr16
mov_l     (vr17), vr14
```

Store 17 in p.x (x is at offset 8 in p)

Struct type (example)

```
int read_i32(void);
void strcpy(char *dst,
            const char *src);

struct Player {
  char name[7];
  int x, y;
  char symbol;
};

int main(void) {
  struct Player p;
  strcpy(p.name, "Frodo");
  p.x = 17;
  p.y = 42;
  p.symbol = '@';
  return 0;
}
```

```
/* high-level IR */
mov_l     vr18, $42
localaddr vr19, $0
mov_q     vr20, $12
add_q     vr21, vr19, vr20
mov_l     (vr21), vr18
```

Store 42 in `p.y` (y is at offset 12 in p)

Struct type (example)

```
int read_i32(void);
void strcpy(char *dst,
            const char *src);

struct Player {
  char name[7];
  int x, y;
  char symbol;
};

int main(void) {
  struct Player p;
  strcpy(p.name, "Frodo");
  p.x = 17;
  p.y = 42;
  p.symbol = '@';
  return 0;
}
```

```
/* high-level IR */
mov_b     vr22, $64
localaddr vr23, $0
mov_q     vr24, $16
add_q     vr25, vr23, vr24
mov_b     (vr25), vr22
```

Store 64 (ASCII code of `'@'` in `p.symbol` (`symbol` is at offset 16 in p)

Explicit address computation vs. address mode (again)

As with array subscript references, code generated for field references in which the field's address is computed explicitly can be rewritten by the peephole optimizer. E.g., the code for `p.x = 17;`:

```
/* before peephole optimization */
leaq      -24(%rbp), %r10
movq      %r10, %r9
movq      %r9, %r10
addq      $8, %r10
movq      %r10, %r8
movl      $17, (%r8)
```

```
/* after peephole optimization */
leaq      -24(%rbp), %r9
movl      $17, 8(%r9)
```