Lecture 10: Arrays and structs

David Hovemeyer

September 18, 2024

601.229 Computer Systems Fundamentals

Example code

All of today's example code linked from course web page as `arraystruct.zip`

Arrays

One-dimensional arrays in C

- Array: sequence of *elements*
  - Each element is just a variable
  - All elements have the same type, the *element type*
  - Number of elements is fixed at time of array creation
- Elements are accessed with an integer index
  - 0 is first element, 1 is second element, etc.
- Subscript operator: `a[i]` refers to the element at index `i` in array `a`

Arrays and pointers in C

- Essential requirement of array element: program must be able to determine its *address*
- For an array, the program just needs to know the *base address* (address of first element)
  - All elements are at a fixed offset from the base address
  - Thus, the address of any element can be `computed` from the base address
  - Address of an element is *base address* + *offset*
- At the machine level, addresses correspond to bytes, so to compute the correct element offset, the array index must be multiplied by the element size in bytes

Example C program

Code:

```
#include <stdio.h>

int main(void) {
  int arr[3] = { 1, 2, 3 };
  printf("%p\n%p\n%p\n", &arr[0], &arr[1], &arr[2]);
  return 0;
}
```

Running the program:

```
$ gcc arrptr.c
$ ./a.out
0x7ffc822662fc
0x7ffc82266300
0x7ffc82266304
```

Example C program

Code:

```
#include <stdio.h>

int main(void) {
  int arr[3] = { 1, 2, 3 };
  printf("%p\n%p\n%p\n", &arr[0], &arr[1], &arr[2]);
  return 0;
}
```

Running the program:

```
$ gcc arrptr.c
$ ./a.out
0x7ffc822662fc
0x7ffc82266300
0x7ffc82266304
```

Note that `sizeof(int)` = 4, and array elements have addresses which differ by 4

Pointer arithmetic

Array/pointer duality:

- If `a` is the name of an array, `a` can also be considered to be a pointer to the first element of the array (i.e., the base address)

Array/pointer identities:

- `a[i]` means the same thing as `*(a + i)`
- This implies that `&a[i]` means the same thing as `(a + i)`
- In general, if `p` points to an array element
  - `p + i` points to the element `i` positions past the one `p` points to
  - `p - i` points to the element `i` positions before the one `p` points to

Pointer difference

- Say that `p` and `q` are pointers, `i` is an integer, and `p` + `i` = `q`
- Then it is also true that `q - p` = `i`
- There is a signed type called `ptrdiff_t` to represent the difference between pointer values
  - It must be a signed type since the difference could be negative
- C language standard only guarantees pointer difference is meaningful when comparing pointers from the same chunk of memory (array, `malloc`'ed buffer, etc.)

Clicker quiz!

Clicker quiz omitted from public slides

Clicker quiz!

Clicker quiz omitted from public slides

Accessing array elements

Goal: write a C function to compute the sum of an array of `uint32_t` elements

Two approaches:

- Array subscript operator
- Use pointer as iterator

First approach

```
uint32_t sum_elts(uint32_t arr[], unsigned len) {
  uint32_t sum = 0;
  for (unsigned i = 0; i < len; i++) {
    sum += arr[i];
  }
  return sum;
}
```

Second approach

```
uint32_t sum_elts(uint32_t arr[], unsigned len) {
  uint32_t *p = arr, *end = arr + len;
  uint32_t sum = 0;
  while (p < end) {
    sum += *p;
    p++;
  }
  return sum;
}
```

Arrays in x86-64 assembly

- Arrays are fairly straightforward to work with in x86-64 assembly
  - Especially if elements are 1, 2, 4, or 8 bytes in size, allowing indexed/scaled addressing
- Any general purpose register can store an address (base address or element pointer)
- Any general purpose register can be used as an index

sum_elts in assembly language

Two implementations of the `sum_elts` function (C versions shown earlier)

C function prototype:

```
uint32_t sum_elts(uint32_t arr[], unsigned len);
```

Recall that in C, an array parameter is really a pointer to the first element of the argument array

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:            <-- initially, %rdi is base addr, %esi is # elements
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax     <-- initialize sum in %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d     <-- use %r10d as index

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d  <-- see if index < n
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone <-- if not, done with loop
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax <-- add arr[index] to sum
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d          <-- increment index
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop      <-- continue loop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, indexed version

```
sum_elts:
    movl $0, %eax
    movl $0, %r10d

.LsumLoop:
    cmpl %esi, %r10d
    jae .LsumLoopDone
    addl (%rdi,%r10,4), %eax
    incl %r10d
    jmp .LsumLoop

.LsumLoopDone:
    ret               <-- sum is in %eax
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:               <-- initially, %rdi is base addr, %esi is # elements
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax      <-- initialize sum in %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10 <-- set %r10 as address past last element

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi   <-- has %rdi gone past last element?
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone <-- if so, done with loop
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax <-- add current element to sum
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi      <-- advance to next element
    jmp .LsumLoop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop        <-- continue loop

.LsumLoopDone:
    ret
```

Sum uint32_t elements, element pointer version

```
sum_elts:
    movl $0, %eax
    leaq (%rdi,%rsi,4), %r10

.LsumLoop:
    cmpq %r10, %rdi
    jae .LsumLoopDone
    addl (%rdi), %eax
    addq $4, %rdi
    jmp .LsumLoop

.LsumLoopDone:
    ret                 <-- sum is in %eax
```

Which approach is better?

- If element size is 1, 2, 4, or 8, then either approach is fine
- Otherwise, the element pointer approach may be preferable (since indexed/scaled addressing can't be used as easily)

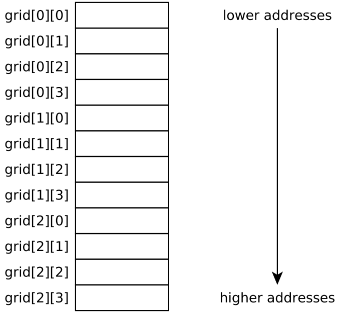
Multidimensional arrays

- Multidimensional arrays in C are laid out in *row-major* order
- Example 2-D array:
  `int grid[3][4];`
- By convention, first dimension is considered "rows", second dimension is considered "columns"

Array structure:

	0	1	2	3
0				
1				
2				

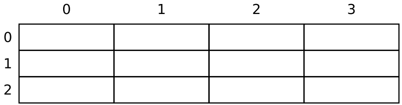
Allocation of elements in memory:



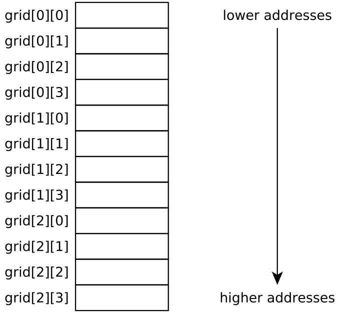
Multidimensional arrays

- Multidimensional arrays in C are laid out in *row-major* order
- Example 2-D array:
  `int grid[3][4];`
- By convention, first dimension is considered "rows", second dimension is considered "columns"

Array structure:



Allocation of elements in memory:



Observation: elements within each row are sequential in memory

Accessing elements of multidimensional arrays

Typical loop to iterate over elements of 2-D array:

```
for (int i = 0; i < NROWS; i++) {
  for (int j = 0; j < NCOLS; j++) {
    /* do something with arr[i][j] */
  }
}
```

Strategy to access elements:

- Each iteration of outer loop computes address of first element of row (`arr[i][0]`)
- The inner loop can then treat `arr[i]` as a one-dimensional array
- Various optimizations are possible
  - For example, loop above accesses elements sequentially in memory, could treat the 2-D array as a 1-D array with `NROWS` × `NCOLS` elements

Structs

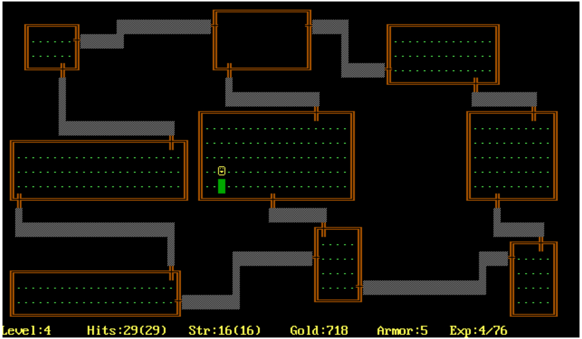
Structs

- A struct (a.k.a. "record") is a *heterogenous* data type consisting of an arbitrary number of fields with arbitrary types
- To access a field within a struct instance, need to know
  - the base address of the struct instance
  - the *offset* of the field being accessed
- Accessing a field is similar to accessing an array element, except that each field has a specific constant offset known at compile time

Struct example

Implementing a game:

```
struct Player {
   int x, y;
   char symbol;
   short health;
};
```



Investigating struct layout

```
#include <stdio.h>

struct Player {
  int x, y;
  char symbol;
  short health;
};

int main(void) {
  printf("%lu\n", sizeof(struct Player));
  return 0;
}
```

Investigating struct layout

```
#include <stdio.h>

struct Player {
  int x, y;
  char symbol;
  short health;
};

int main(void) {
  printf("%lu\n", sizeof(struct Player));
  return 0;
}
```

What output does this program print?

Output of program

```
$ gcc structlayout.c
$ ./a.out
12
```

Output of program

```
$ gcc structlayout.c
$ ./a.out
12
```

Why 12?

Alignment and padding

- Compiler must ensure that memory accesses are properly aligned
  - E.g., a 4 byte `int` variable must have its storage allocated at an address that is a multiple of 4
- When laying out the fields of a struct type, the compiler may need to add padding before or after fields to ensure correct alignment

Clicker quiz!

Clicker quiz omitted from public slides

Investigating field offsets

```
#include <stdio.h>

#define OFFSET_OF(s,f) \
((unsigned) ((char*)&s.f - (char*)&s))

struct Player {
  int x, y;
  char symbol;
  short health;
};

int main(void) {
  struct Player p;
  printf("x offset=%u\n", OFFSET_OF(p,x));
  printf("y offset=%u\n", OFFSET_OF(p,y));
  printf("symbol offset=%u\n", OFFSET_OF(p,symbol));
  printf("health offset=%u\n", OFFSET_OF(p,health));
  return 0;
}
```

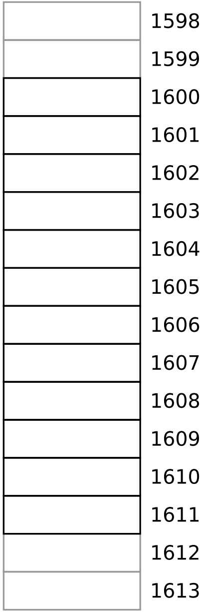
Running the program

```
$ gcc structlayout3.c
$ ./a.out
x offset=0
y offset=4
symbol offset=8
health offset=10
```

Visualizing struct layout

```
struct Player {
  int x, y;
  char symbol;
  short health;
};
```

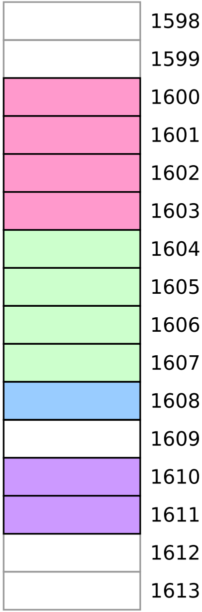
Assume that the base address of an instance of `struct Player` is 1600



Visualizing struct layout

```
struct Player {
  int x, y;
  char symbol;
  short health;
};
```

Assume that the base address of an instance of `struct Player` is 1600

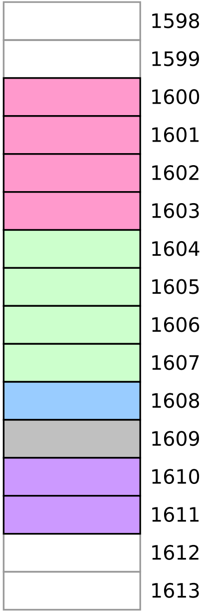


Field layout

Visualizing struct layout

```
struct Player {
  int x, y;
  char symbol;
  short health;
};
```

Assume that the base address of an instance of `struct Player` is 1600



Byte at offset 9 is padding

Clicker quiz!

Clicker quiz omitted from public slides

Ensuring field alignment

The compiler may need to add padding at the end of the struct to guarantee alignment of all fields

For example, if the struct type has fields requiring 8 bytes, the size of the struct must be a multiple of 8

Accessing struct fields in assembly language

- Accessing struct fields in assembly language is super easy!
- Assuming that you have the base address of a struct instance in a register, the field is at a fixed offset from the base address
- Specify that offset when making the memory reference

C example

```
struct Player {
  int x,          // offset 0
      y;          // offset 4
  char symbol;
  short health;
};

void move_player(struct Player *p, int dx, int dy) {
  p->x += dx;
  p->y += dy;
}
```

Assembly example

```
/* Note that first three arguments are in %rdi, %rsi, and %rdx */

move_player:
    addl %esi, 0(%rdi)          /* p->x += dx */
    addl %edx, 4(%rdi)          /* p->y += dy */
    ret
```

Even better assembly example

```
/* Note that first three arguments are in %rdi, %rsi, and %rdx */

#define PLAYER_X_OFFSET 0
#define PLAYER_Y_OFFSET 4

move_player:
    addl %esi, PLAYER_X_OFFSET(%rdi)  /* p->x += dx */
    addl %edx, PLAYER_Y_OFFSET(%rdi)  /* p->y += dy */
    ret
```