Structs

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Meet struct
- Accessing and initializing structs
- Pass-by-address and pass-by-value
- Literal struct values
- Nesting structs and arrays
- Meet typedef

Thinking about Structs

- A struct is like a class in Java
  - It aggregates multiple fields, each with its own name and type
  - Unlike Java, a struct only contains data, no methods
  - Unlike Java, all the fields of a struct are accessible to any code (it's as if they are all public)

```
struct Person {
    char name[ 12 ];
    double height;
    int age;
};
```

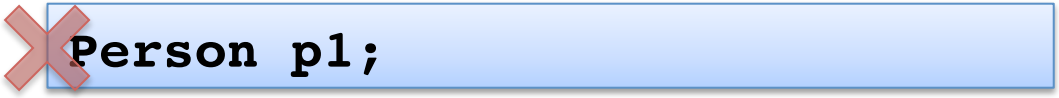
Structure name

Fields in the struct (you can't initialize these here)

It's easy to forget this semi-colon.

Using Structs

- You can create variables with your new structure type.
- But, defining a new struct doesn't automatically introduce a new type name.



```
Person p1;
```

- It introduces a structure name …
  - but you have to use struct keyword to talk about the structure's type:

```
struct Person p1;
```

Using Structs

- Once you've made an instance, you can access its fields using the dot operator (just like in Java)

```
p1.height = 1.75;
p1.age = 24;


printf( "%d\n", p1.age );
double x = p1.height;


p1.name = "Mary";
```

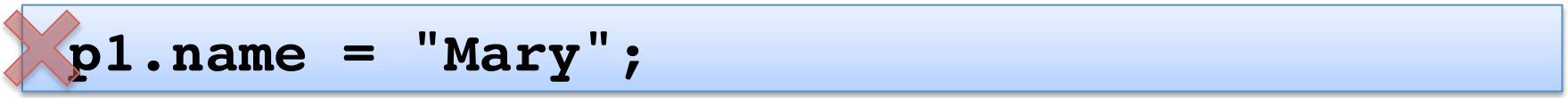
You can store values in the fields.

You can get values from the fields.

But you can't do this. It's not a struct problem; you just can't assign between arrays like this.

Using Structs

- So, you can't assign to a string field like this:



```
p1.name = "Mary";
```

- How could you initialize that name field?
  - You could copy the string yourself, one character at a time.
  - You could read something into it, using scanf()
  - You could use a library routine, like strcpy()
  - Or …

Structure Initialization

- You can initialize a struct instance's fields when you declare it:

```
struct Person p2 = { "William", 1.85, 27 };
```

In the same order as the struct definition.

This will do a deep copy of this string.

- C99 Introduced a new, more explicit syntax:

```
struct Person p3 = { .age = 33,
                     .name = "Agatha",
                     .height = 1.7 };
```

Structure Initialization

- As with arrays, storage class determines how a structs fields will be initialized.

```
struct Person p4;
printf( "age: %d\n", p4.age );
```

I wonder what this will print. Nobody knows.

- If you use initialization syntax, any fields you don't specify get initialized to zero.

```
struct Person p4 = { .height = 1.234 };
printf( "age: %d\n", p4.age );

struct Person p5 = { "abc" };
struct Person p6 = {};
```

OK. This field will be zero.

How about these?

Combined Definition and Declaration

- You can declare struct variables along with the struct definition.

```
struct Grade {
  double minimum;
  char letter;
} aGrade = { 90.0, 'A' };
```

See. That's why we need this semicolon.

- You don't even need to give the structure a name, if you'll never use it again.

```
struct {
  double latitude;
  double longitude;
} raleigh = { 35.7806, 78.6389 };
```

Thinking about Structs

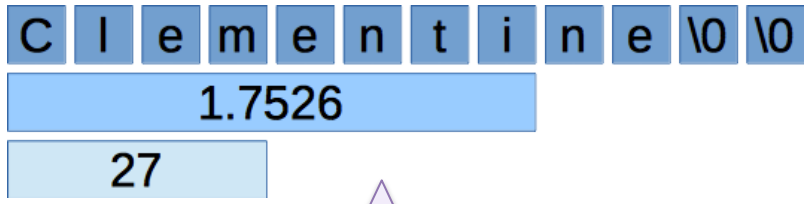
- You can think of an array like this:

```
short list[] = { … };
```

- You can think of a struct like this:

```
struct Person {
    char name[ 12 ];
    double height;
    int age;
};

struct Person p1 = { … };
```



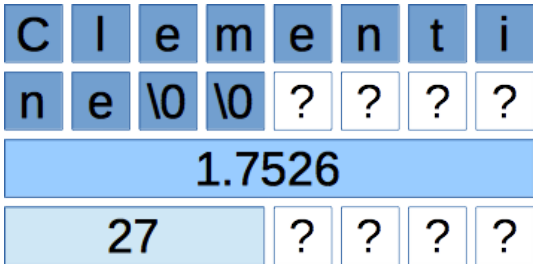
Fields of different sizes, one after another in memory.

Struct Layout

- But, your struct fields may not be laid out exactly like you expect.
  - They will be in the order you specify
  - … but, the compiler may introduce *padding* between some fields.
  - Why? Performance, and the hardware may have *alignment requirements.*

```
struct Person {
    char name[ 12 ];
    double height;
    int age;
};

struct Person p1 = { … };
```

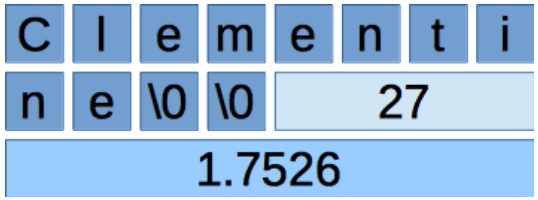


Struct Layout

- On some systems, ints may need to be *2-byte aligned* … starting on an even memory address.
- Or, they may need to be *4-byte aligned*.
- More generally, *natural alignment* requires a starting address that's a multiple of the value's size.
- Reorganizing the structure in memory may let the compiler lay it out more efficiently.

```
struct Person {
    char name[ 12 ];
    int age;
    double height;
};

struct Person p1 = { … };
```



Structs and Value Semantics

- Structs support *value semantics*
  - You can assign from one instance to another and you get a deep copy
  - You can pass and return by value when you call a function.

```
struct Event {
  char name[ 10 ];
  int hour;
  int minute;
};
```

```
  struct Event e1 = { "Wake Up", 6, 30 };
  struct Event e2;

  e2 = e1;

  if ( e2 == e1 )
    printf( "They're the same!\n" );
```

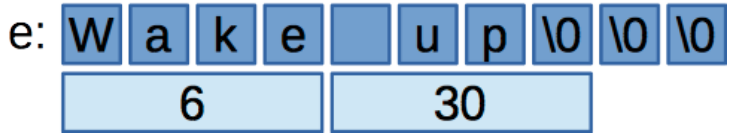
Now e2 contains a copy of e1.

But you can't compare values like this.

Structs and Pass-By-Value

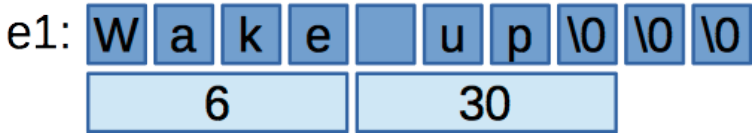
- So, we can define a function like this to print an event:

```
void printEvent( struct Event e )
{
  printf( "%s %d:%02d\n", e.name,
          e.hour, e.minute );
}
```



- We can call it, passing in a copy of the event struct.

```
  printEvent( e1 );
```

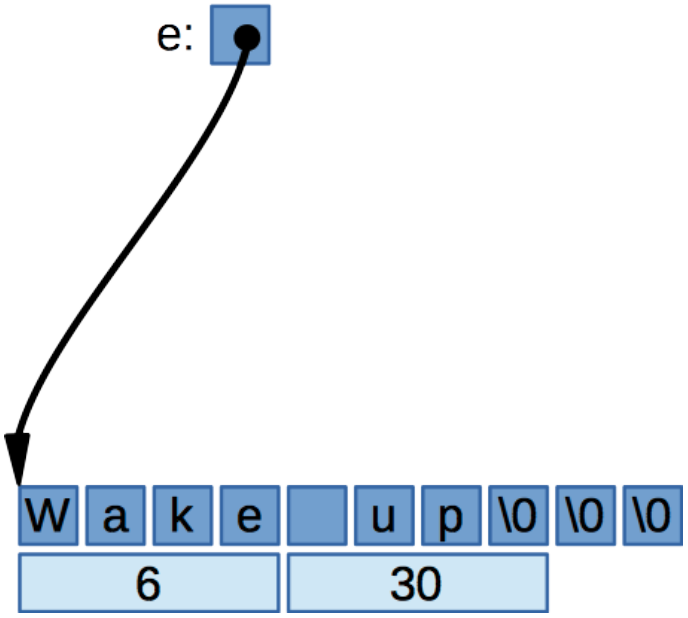


Structs and Pass-By-Reference

- We can use pointers to pass the struct by reference instead.

```
void printEvent( struct Event const *e )
{
  printf( "%s %d:%02d\n", (*e).name,
          (*e).hour, (*e).minute );
}
```

Do we need these parentheses?



- We just need to pass the struct's address.

```
  printEvent( &e1 );
```

Field via Pointer

- We usually end up passing around pointers to struct much more often than passing copies of struct instances.
  - So, we end up writing syntax like the following a lot: **`(*ptr).field`**
  - Or, we would if we didn't have a shorthand: **`ptr->field`**
- This would let us write the printEvent function like:

```
void printEvent( struct Event const *e )
{
  printf( "%s %d:%02d\n", e->name, e->hour, e->minute );
}
```

Structs and Precedence

- Our two new operators are in the highest precedence category.

Operator	Description
++ -- () [] **.** **->**	Postincrement, postdecrement : e.g., a++ Function call : e.g., f(5) Array index : e.g., a[5] **Field from struct value: e.g., b.x** **Field from struct pointer: e.g., bptr->x**
++ -- + - ! (*type*) * & sizeof	Preincrement, predecrement : e.g., --a Unary positive, negative : e.g., -a Logical not: e.g., !a Type cast: e.g., (float) a Pointer dereference: e.g., *p Address-of : e.g., &a Sizeof operator : e.g., sizeof(int)
* / %	Multiply, divide, mod

Returning Structs

- Functions can return structs as their return values.

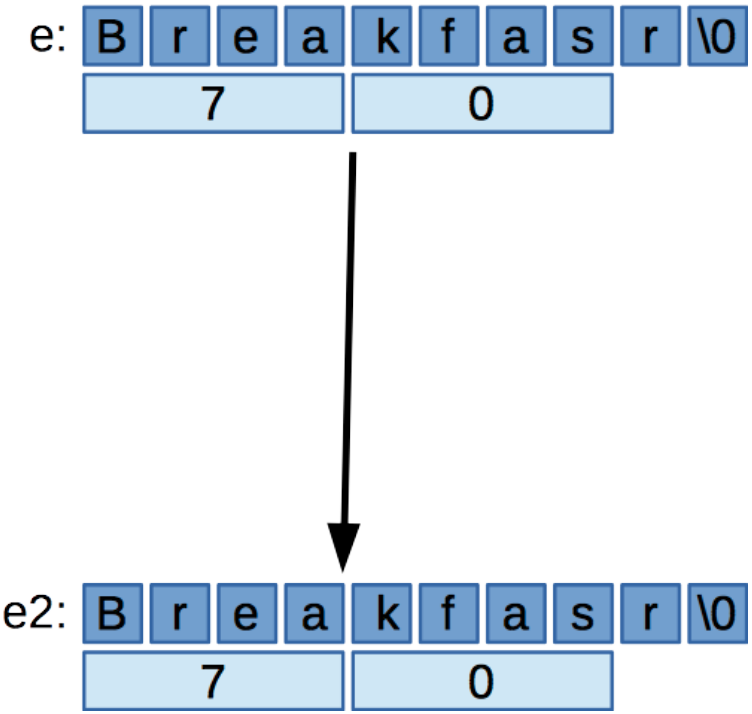
```
struct Event readEvent()
{
  struct Event e;

  printf( "Event Name: " );
  scanf( "%s", e.name );

  printf( "Time: " );
  scanf("%d:%d", &e.hour, &e.minute);

  return e;
}
```

Do we need parentheses here?




```
e2 = readEvent();
```

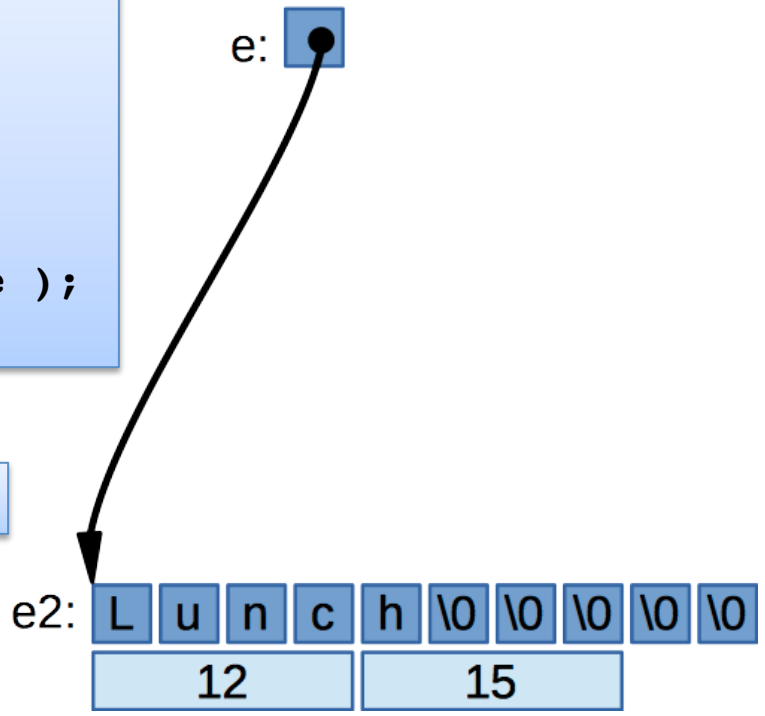
Modifying Reference Parameters

- Or, we could pass the struct by address and let the function modify it directly.

```
void readEvent( struct Event *e )
{
  printf( "Event Name: " );
  scanf( "%s", e->name );

  printf( "Time: " );
  scanf( "%d:%d", &e->hour, &e->minute );
}
```

```
  readEvent( &e2 );
```



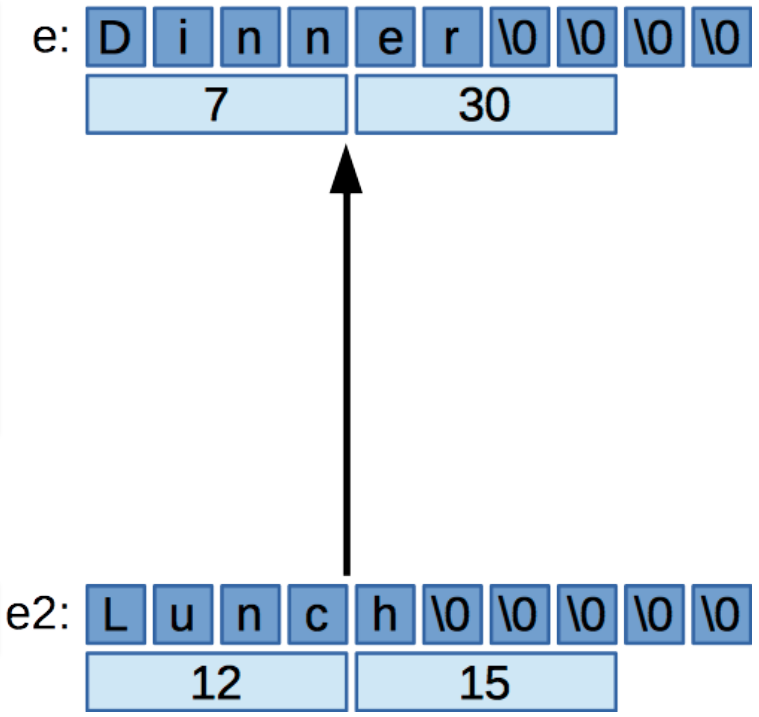
Bad Ideas

- We can pass by value.
  - But here the function is just changing a copy of the caller's struct.

```
void readEvent( struct Event e )
{
  printf( "Event Name: " );
  scanf( "%s", e.name );

  printf( "Time: " );
  scanf( "%d:%d", &e.hour, &e.minute );
}
```

```
readEvent( e2 );
```



Bad Ideas

- We can return by address.
  - But here we're returning the address of a variable that's going away.

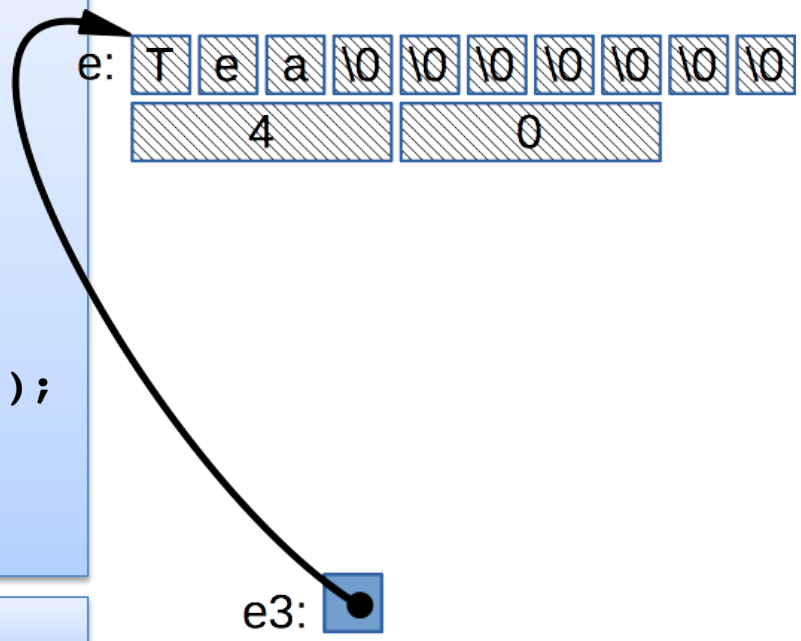
```
struct Event *readEvent()
{
  struct Event e;

  printf( "Event Name: " );
  scanf( "%s", e.name );

  printf( "Time: " );
  scanf( "%d:%d", &e.hour, &e.minute );

  return &e;
}
```

```
struct Event *e3 = readEvent();
```



Dynamically Allocating Structs

- A struct instance occupies a contiguous block of memory.
  - That memory can come from any of our three storage regions, static, stack … or heap.
- We just need to ask for the right amount.
- Here's how:

```
struct Event *e =
     (struct Event *)malloc( sizeof( struct Event ) );
```

Returning a New Struct

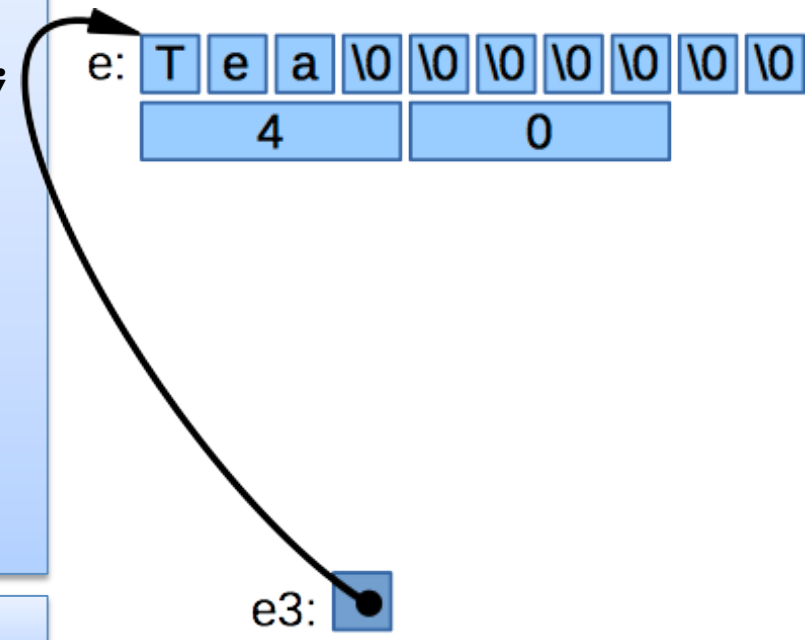
- With dynamic allocation, we can fix return-by-address.

```
struct Event *readEvent()
{
  struct Event *e = (struct Event *)
    malloc( sizeof( struct Event ) );

  printf( "Event Name: " );
  scanf( "%s", e->name );
  printf( "Time: " );
  scanf( "%d:%d", &(e->hour),
         &(e->minute) );

  return e;
}
```

```
struct Event *e3 = readEvent();
```



Literal Struct Values

- Notice, initialization syntax is special.
- This means make me an array that contains exactly this string:

```
char str1[] = “abc123”;
```

- But this means just take a pointer to a literal string.

```
char *str2 = “abc123”;
```

Literal Struct Values

- During declaration, we have a syntax for specifying initial array and struct values.

```
int list[] = { 12, 15, 35 };
struct Event evt = { "Concert", 8, 0 };
```

- But, outside initialization, this syntax is meaningless (well, at least it doesn't mean what we want).

```
f( { 12, 15, 35 } );
e2 = { "Concert", 8, 0 };
```

Literal Struct Values

- C99 gives us a way to use this syntax.
- It looks like a type cast … but it's not.
- You've already seen this for arrays.

```
f( ( int [] ) { 12, 15, 35 } );
```

- We can use this for structs also.

```
e2 = ( struct Event ) { "Concert", 8, 0 };
```

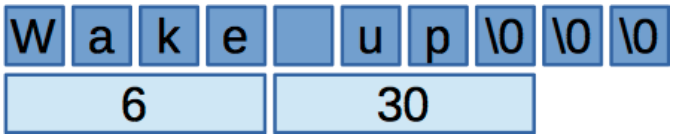
- This is stack-allocated storage. You can take its address but it goes away when you leave the current scope.

Nesting Structures and Arrays

- You've seen, we can nest arrays inside structures.
- Really, you have two choices
- Put an array inside a structure
  - Array stored along with structure fields
  - Array length must be a constant expression (so, all instances have the same array size)

```
struct Event {
  char name[ 10 ];
  int hour;
  int minute;
};
```

```
  char letter = e.name[ 0 ];
```

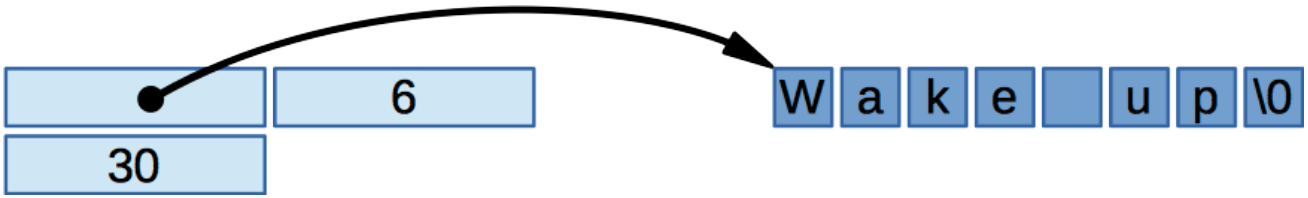


Nesting Structures and Arrays

- Or, store a pointer inside a structure
  - Pointing to the array contents
  - You can still index using the same syntax
  - Array size can vary from instance to instance
  - But, you have to find memory elsewhere to store the array.
  - This is more like how Java does it.

```
struct Event {
  char *name;
  int hour;
  int minute;
};
```

```
  char letter = e.name[ 0 ];
```

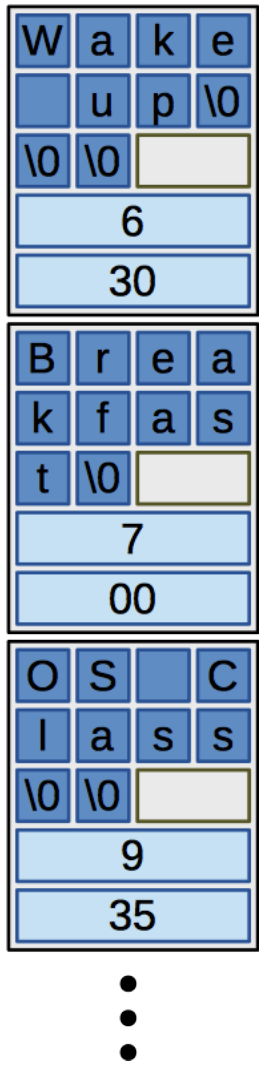


Arrays of Structs

- You can store structures as array elements.

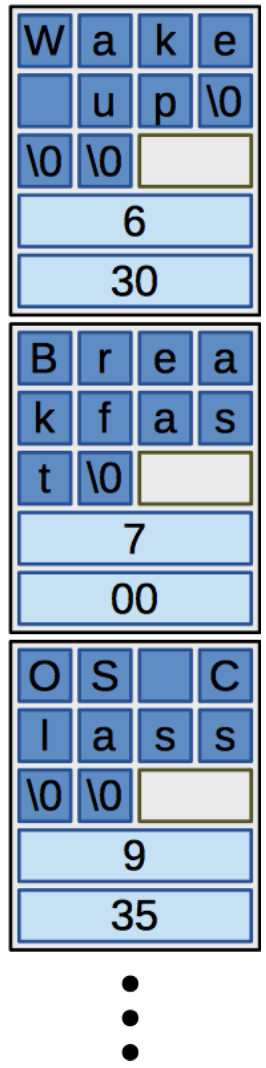
```
struct Event schedule[] = {
  { “Wake up”, 6, 30 },
  { “Breakfast”, 7, 0 },
  { “OS Class”, 9, 35 },
  { .hour = 11, .minute = 0, .name = “Meeting” },
  [ 5 ] = { “C Class”, 11, 45 }
};
```

You can mix the initialization syntax.



Arrays of Struct Instances

```
struct Event schedule[] = {
  { "Wake up", 6, 30 },
  { "Breakfast", 7, 0 },
  { "OS Class", 9, 35 },
  { .hour = 11, .minute = 0, .name = "Meeting" },
  [ 5 ] = { "C Class", 11, 45 }
};
```



- To get to an instance, we index into the array
- To get to one of its fields, we use dot.

```
char *ename  = schedule[2].name;
double etime = schedule[3].hour +
               schedule[3].minute/60.0 ;
```

Arrays of Struct Instances

- We've been allocating these arrays on the stack (or, statically).
- Alternatively, we could dynamically allocate them.

Pointer to the start of the array.

Number of items times size of each item.

```
struct Event *schedule =
     (struct Event *) malloc(6 * sizeof(struct Event));
```

This memory will be uninitialized.

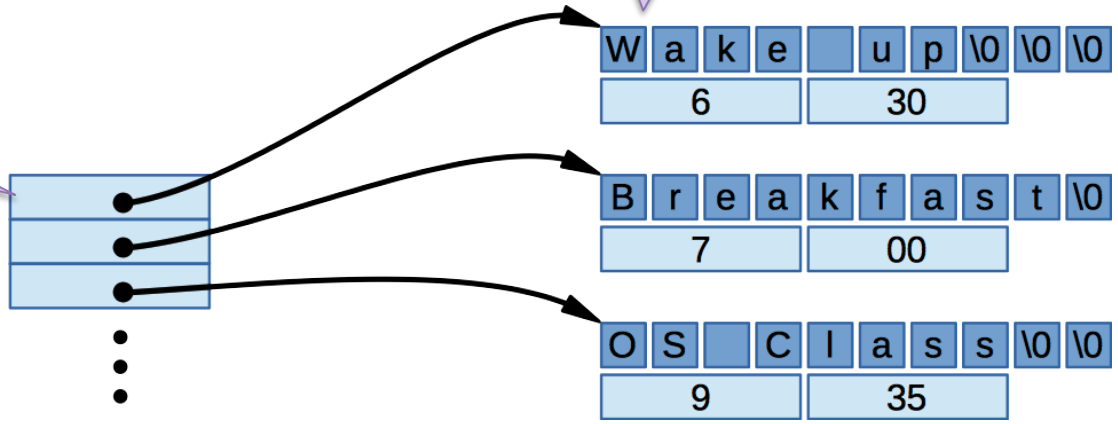
Arrays of Pointers to Structs

- We can make an array of pointers to structs

```
struct Event *schedule[] = {
  &(struct Event){ "Wake up", 6, 30 },
  &(struct Event){ "Breakfast", 7, 0 },
  &(struct Event){ "OS Class", 9, 35 },
  &(struct Event){ "Meeting", 11, 0 },
  &(struct Event){ "C Class", 11, 45 }
};
```

Each struct instance stored elsewhere in memory.

More like what we get in Java.



Arrays of Pointers to Structs

```
struct Event *schedule[] = {
  &(struct Event){ "Wake up", 6, 30 },
  &(struct Event){ "Breakfast", 7, 0 },
  &(struct Event){ "OS Class", 9, 35 },
  &(struct Event){ "Meeting", 11, 0 },
  [ 5 ] = &(struct Event){ "C Class", 11, 45 }
};
```

Just showing you can mix this syntax. Skipped an item.

- To get to a field, we need to index into the array then access the field via a pointer.

```
  char *ename = schedule[ 2 ]->name;
  char letter = schedule[ 3 ]->name[ 0 ];
```

Arrays of Pointers to Structs

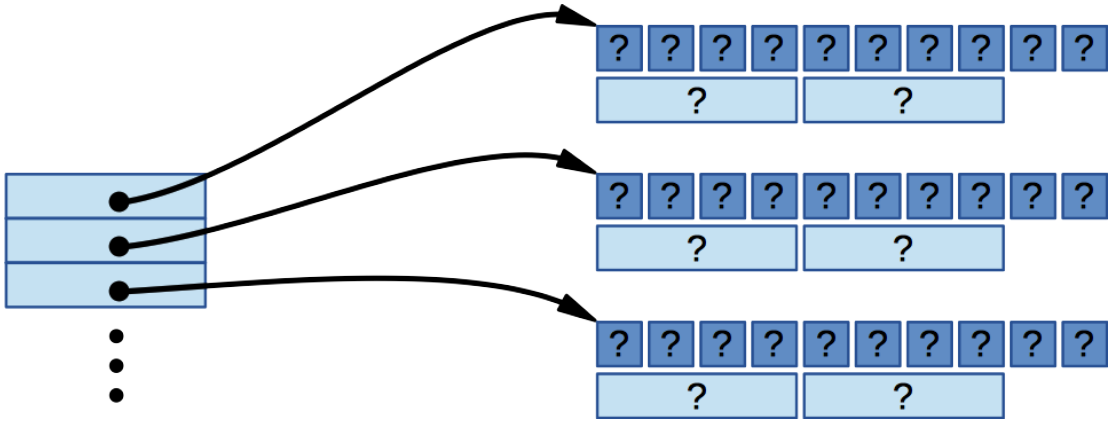
- Here also, we could dynamically allocate

Pointer to the start … of an array of pointers.

Number of pointers times size of each pointer.

```
struct Event **schedule =
    (struct Event **) malloc(6 * sizeof( struct Event*))
for ( int i = 0; i < 6; i++ )
  schedule[ i ] = (struct Event *) malloc(sizeof( struct Event ));
```

Make each of these pointers point to a new instance.

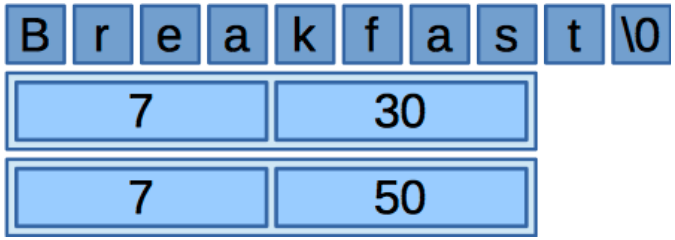


Nesting Structures and Arrays

- You can put a struct inside another struct.

```
struct Time {
  int hour;
  int minute;
};

struct Event {
  char name[ 10 ];
  struct Time start;
  struct Time end;
};
```




```
  struct Event evt = { "Breakfast",
                       { 7, 30 },
                       { 7, 50 }
                     };

  evt.end.minute = 55;
```

A nested structure initialization.

To get to an element, you need a field inside a field.

Meet a New Friend, typedef

- See, we can make long, complicated type names in C … and understand them.
- But we don't always want to; sometimes its nice to have a short name
- And, we can, using `typedef`
  - Use typedef just before something that looks like a variable declaration.
  - You're not declaring a variable, you're introducing a new type name.

```
typedef int Table[ 10 ][ 10 ];
```

```
Table tbl1, tbl2;
```

Why Use typedef?

- Typedef lets us isolate platform dependencies.
- We could put this in one header:

```
typedef long int64;
```

- Everywhere else, we can just use our type name:

```
int64 counter = 0;
```

- If we change platforms, we only need to update the header to reflect platform differences.

Why Use typedef?

- Typedef lets us simplify naming and thinking about complex types.
- Here's a pointer to a function.

```
typedef bool (*TestFunctionPtr)( int );
```

- Here's an array of those pointers.

```
TestFunctionPtr testList[ 10 ];
```

- This can help to reduce the zig-zagging when reading a type name.

Typedef and 2D Arrays

- This can also simplify describing multi-dimensional arrays.

```
typedef int Row[ 20 ];

Row *table;

table = (Row *) malloc( 50 * sizeof( Row ) );
```

I'm a pointer to a Row, so you can use me like an array of Rows.

Now, I'm an array of 50 rows.

- We could do without this typedef, but it makes the types harder to describe.

```
int (*table)[ 20 ];

table = (int (*)[20]) malloc( 50 * 20* sizoef( int ) );
```

Typedef and Structs

- Typedef lets us create type names for structure types.

```
struct EventStruct {
  char name[ 10 ];
  int hour, minute;
};

typedef struct EventStruct Event;

Event evt1 = { "Nap", 2, 5 };
```

It's OK to declare two fields at once.

Now, I can use Event like a regular type name.

- This is a common practice for structs.

Typedef and Structs

- You can even give a type name for an otherwise anonymous struct.

```
typedef struct {
  char name[ 10 ];
  int hour, minute;
} Event;

Event evt1 = { "Nap", 2, 5 };
```

Typedef and struct definition all in one.

- This has some disadvantages, so it's less common.

Typedef Changes Nothing!

- Typedef doesn't create a new type
- It just gives you a new name for a type.
- This is good.

```
typedef struct EventStruct {
  char name[ 10 ];
  int hour, minute;
} Event;

struct EventStruct evt1 = { "Study", 2, 5 };
Event evt2 = { "Nap", 2, 5 };

evt2 = evt1;
```

Declared with two different type names.

But, the compiler knows they are the same.