Intermediate Programming

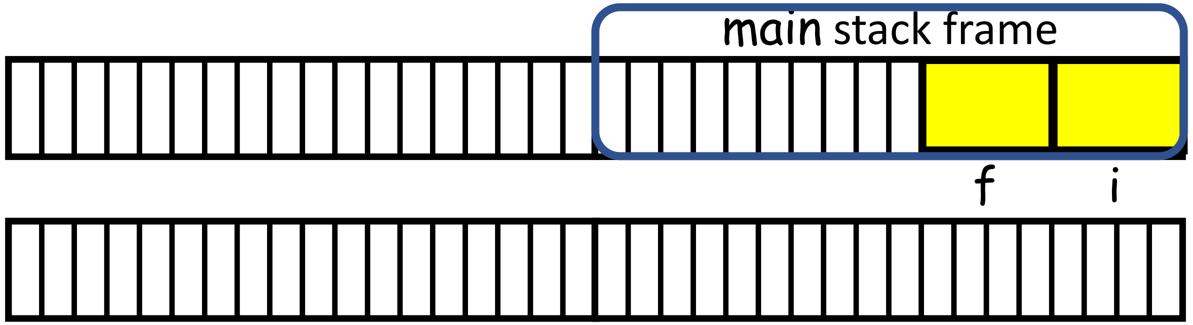
Day 13

Outline

- Lifetime and scope
- `struct`s
- `typedef`
- Random numbers
- Review questions

Variable lifetime and scope

- Variables declared in C programs have:
  - *lifetime*: How long is the variable in memory?
    - Both f and i have a lifetime equal to the duration of the main function
      (They come into existence when main's stack frame is created and disappear when it's gone)
  - *scope*: Where is the variable name accessible?
    - f is in scope from the point it is declared to the end of the main function (lines 4-7)
    - i is in scope for the for loop (lines 5-6)




```
1. #include <stdio.h>
2. int main( void )
3. {
4.        int f= 1;
5.        for( int i=2 ; i<6 ; i++ )
6.               f*= i;
7.        printf( "%d\n" , f);
8. }
```

Variable lifetime and scope

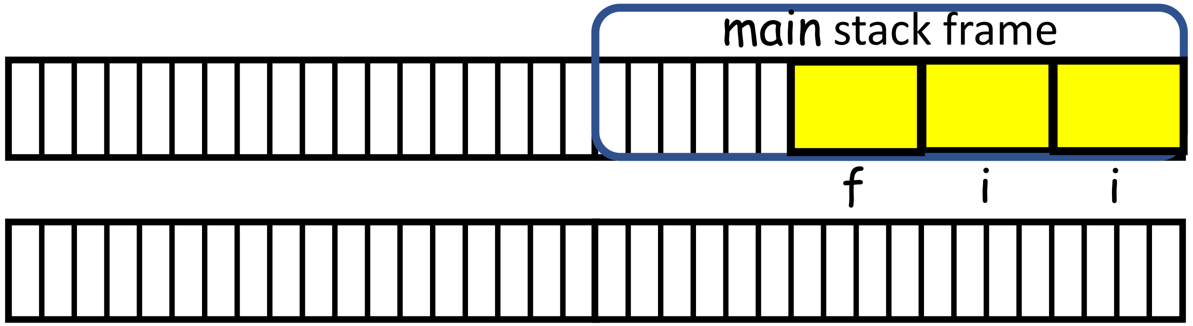
Q: What are the lifetimes of the variables `i`?

A: Both have a lifetime equal to the duration of the `main` function

Q: What are the scopes of the variables `i`?

A: The first comes into scope when it is declared, is shadowed / hidden during the `for` loop, and re-emerges after (lines 4, 7)

The second is in scope during the `for` loop (lines 5-6)




```
1. #include <stdio.h>
2. int main( void )
3. {
4.        int i,f=1;
5.        for( int i=2 ; i<6 ; i++ )
6.                f*= i;
7.        printf( "%d\n" , f);
8. }
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't shadowed / hidden by an inner variable with the same name)

```
#include <stdio.h>
void foo( int i )
{
    static int count;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 ) ;
    foo( 7 );
    return 0;
}
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't shadowed / hidden by an inner variable with the same name)
  - But… prefixing the variable declaration with the `static` keyword, extends the lifetime across all calls to that function
    - The variable is automatically initialized to have zero value

```
#include <stdio.h>
void foo( int i )
{
    static int count;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 ) ;
    foo( 7 );
    return 0;
}
```

```
>> ./a.out
0] foo( 1 )
1] foo( 7 )
>>
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't shadowed / hidden by an inner variable with the same name)
  - But… prefixing the variable declaration with the `static` keyword, extends the lifetime across <u>all</u> calls to that function
    - The variable is automatically initialized to have zero value
    - If you declare and assign, the assignment only happens the first time the function is called.

```
#include <stdio.h>
void foo( int i )
{
    static int count=5;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 ) ;
    foo( 7 );
    return 0;
}
```

```
>> ./a.out
5] foo( 1 )
6] foo( 7 )
>>
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't shadowed / hidden by an inner variable with the same name)
  - But… prefixing the variable declaration with the `static` keyword, extends the lifetime across <u>all</u> calls to that function
  - But the variable is still only scoped within the function

```
#include <stdio.h>
void foo( int i )
{
    static int count=5;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

Variable lifetime and scope

Note:
Because a static variable's lifespan extends beyond the function call, it does not reside on the stack.
(static variables are stored in the *data segment*.)

- Variabl
  - In ge ... n's duration (ass hidd the same name)
  - But… prefixing the variable declaration with the static keyword, extends the lifetime across all calls to that function
  - But the variable is still only scoped within the function

```
{
    static int count=5;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - We can also define *global* variables outside of any function
    - They have a lifetime equal to the lifetime of the program
      - They are initialized to zero
    - They are accessible to any function following the declaration

```
#include <stdio.h>
int count;
void foo( int i )
{
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

```
>> ./a.out
0] foo( 1 )
1
>>
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - We can also define *global* variables outside of any function
    - They have a lifetime equal to the lifetime of the program
      - They are initialized to zero
    - They are accessible to any function following the declaration

```
#include <stdio.h>
int count;
void foo( int i )
{
    printf( "%d] foo( %d )\n" , count++ ,  i );
}

main( void )

    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

```
>> ./a.out
0] foo( 1 )
1
>>
```

<u>Note</u>:
Like `static` variables, global variables do not reside on the stack.
(They too are stored in the *data segment*.)

Variable lifetime and scope

Global variables:

- Like functions, you can define global variables in one source file and use them in another.
- At compile time, the compiler only needs to know the declaration, not the definition.
- At link time, the linker will bind the declared variables to their definitions.

```
int count = 3;
```

foo.c

```
#include <stdio.h>

void incrementCount( int i )
{
    extern int count;
    count += i;
}
int main( void )
{
    extern int count;
    incrementCount( 5 );
    printf( "%d\n" , count );
    return 0;
}
```

main.c

Variable lifetime and scope

Global variables:

- Like functions, you can define global variables in one source file and use them in another.
- At compile time, the compiler only needs to know the declaration, not the definition.
- At link time, the linker will bind the declared variables to their definitions.
- The `extern` keyword can be used to declare global variables that are defined elsewhere (either in the same file or in other files).

```c
int count = 3;
```

foo.c

```c
#include <stdio.h>

void incrementCount( int i )
{
    extern int count;
    count += i;
}
int main( void )
{
    extern int count;
    incrementCount( 5 );
    printf( "%d\n" , count );
    return 0;
}
```

main.c

```
>> gcc main.c foo.c ...
>> ./a.out
8
>>
```

Variable lifetime and scope

You can also declare the variable outside of a function call so that all (subsequent) functions calls have access to it.

needs to know the declaration, not the definition.

- At link time, the linker will bind the declared variables to their definitions.
- The `extern` keyword can be used to declare global variables that are defined elsewhere (either in the same file or in other files).

```
int count = 3;
```

foo.c

```
#include <stdio.h>
extern int count;

void incrementCount( int i )
{
    count += i;
}
int main( void )
{
    incrementCount( 5 );
    printf( "%d\n" , count );
    return 0;
}
```

main.c

```
>> gcc main.c foo.c ...
>> ./a.out
8
>>
```

Beware the global variable

- Usage of global variables is generally discouraged
  - Debugging is harder – less clear which function changed a global variable's value (since it could be any!)
  - Global variables cross boundaries between program modules, undoing benefits of modular code
    - readability
    - testability
  - In general, values should be conveyed via parameter passing and return values

Outline

- Lifetime and scope
- structs
- typedef
- Random numbers
- Review questions

Structures (`struct`s)

- If we have an application that stores students' ages and grades, we can represent a student's data by an array of `float` values. (E.g. by storing the data for N students in a `float` array of size 2N.)

Q: What if we want to store other (non-numerical) data like names?

A: A structure is a collection of variables (often heterogeneously-typed) that are bundled together as a unit under a single name

Structures (**struct**s)

- Use the **struct** keyword to define a new type

```
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
```

Structures (**struct**s)

- Use the **struct** keyword to define a new type
  - It has a (type) name

```
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
```

Structures (**struct**s)

- Use the **struct** keyword to define a new type
  - It has a (type) name
  - And a list of variables (members)

```
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
```

Structures (**struct**s)

- Use the `struct` keyword to define a new type
  - It has a (type) name
  - And a list of variables (members)

- Variables of the type are declared using the `struct` keyword and the `struct` (type) name

```
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};

struct Rec boss;
struct Rec assistant;
```

Structures (**structs**)

- Use the **struct** keyword to define a new type
  - It has a (type) name
  - And a list of variables (members)

- Variables of the type are declared using the **struct** keyword and the **struct** (type) name
  - Can initialize members using array syntax
    - Variables must be in the same order

```
boss = { 1 , "misha" , 0.f };
```

```
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};

struct Rec boss;
struct Rec assistant;
```

Structures (**structs**)

- Use the **struct** keyword to define a new type
  - It has a (type) name
  - And a list of variables (members)

- Variables of the type are declared using the **struct** keyword and the **struct** (type) name
  - Can initialize members using array syntax
  - Or member-by-member, using the "." operator

```
boss = { 1 , "misha" , 0.f };
```

```
boss.eNum = 1;
boss.name = "misha";
boss.salary = 0.f;
```

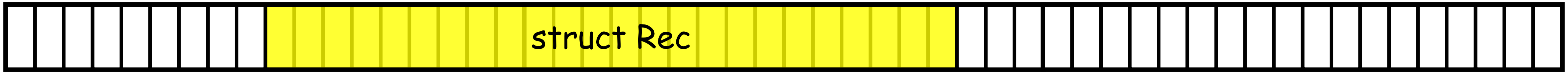
```
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};

struct Rec boss;
struct Rec assistant;
```

Structures (**struct**s)

- When the compiler sees a **struct** type it creates enough memory on the stack to store all of its contents

```c
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec rec;
    ...
    return 0;
}
```

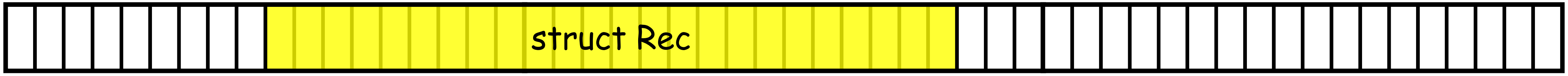


address space

Structures (**struct**s)

- When the compiler sees a **struct** type it creates enough memory on the stack to store all of its contents
    - You can get the size of the memory associated to a struct using **sizeof** …

```c
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec rec;
	printf( "Size: %d\n" ,
		(int)sizeof( rec ) );
	return 0;
}
```



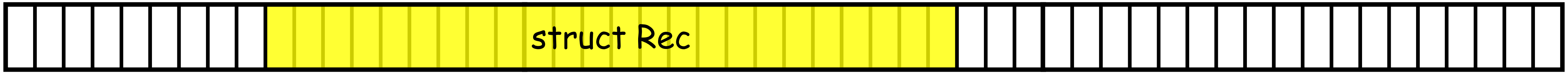
address space

Structures (**structs**)

- When the compiler sees a **struct** type it creates enough memory on the stack to store all of its contents
  - You can get the size of the memory associated to a struct using **sizeof** … but this might be larger than the sum of its parts

```c
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec rec;
    printf( "Size: %d\n" ,
        (int)sizeof( rec ) );
    return 0;
}
```

```
>> ./a.out
Size: 24
>>
```



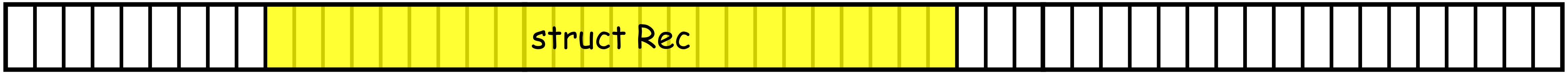
address space

Structures (**structs**)

- When the compiler sees a **struct** typ
  creates enough memory on the stack
  store all of its contents
  - You can get the size of the memory asso
    a struct using **sizeof** … but this might be
    than the sum of its parts

```c
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	printf( "%d + " , sizeof( unsigned int ) );
	printf( "%d + " , sizeof( const char* ) );
	printf( "%d = " , sizeof( float ) );
	printf( "%d\n" , sizeof( struct Rec ) );
	return 0;
}
```

```
>> ./a.out
4 + 8 + 4 = 24
>>
```



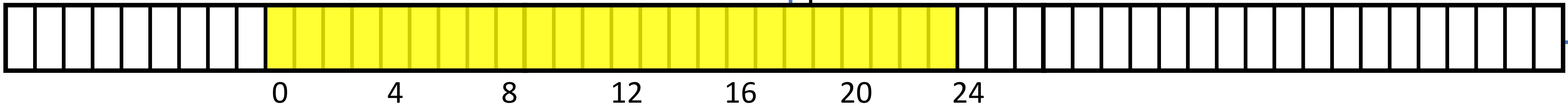
address space

Structures (**structs**)

- When the compiler sees a **struct** …
  creates enough memory on the sta…
  store all of its contents
  - You can get the size of the memory a…
    a struct using **sizeof** … but this migh…
    than the sum of its parts

```
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec r;
	void *_r = &r;
	void *_e = &(r.eNum);
	void *_n = &(r.name);
	void *_s = &(r.salary);
	printf( "Size: %d\n" , sizeof( struct Rec ) );
	printf( "eNum offset: %d\n" , _e - _r );
	printf( "name offset: %d\n" , _n - _r );
	printf( "salary offset: %d\n" , _s - _r );
	return 0;
}
```

```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```

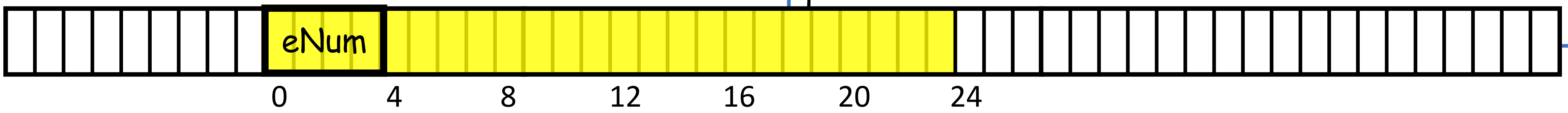


Structures (**structs**)

- When the compiler sees a **struct** …
  creates enough memory on the sta…
  store all of its contents
  - You can get the size of the memory a…
    a struct using **sizeof** … but this migh…
    than the sum of its parts

```
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec r;
	void *_r = &r;
	void *_e = &(r.eNum);
	void *_n = &(r.name);
	void *_s = &(r.salary);
	printf( "Size: %d\n" , sizeof( struct Rec ) );
	printf( "eNum offset: %d\n" , _e - _r );
	printf( "name offset: %d\n" , _n - _r );
	printf( "salary offset: %d\n" , _s - _r );
	return 0;
}
```

```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```

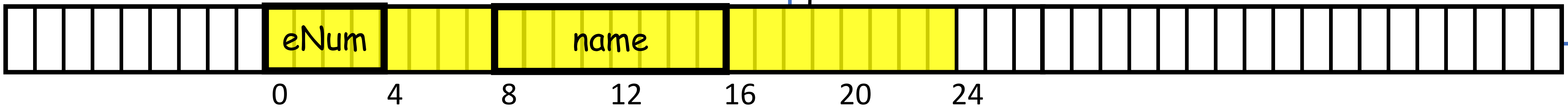


Structures (**structs**)

- When the compiler sees a **struct**
  creates enough memory on the sta
  store all of its contents
  - You can get the size of the memory a
    a struct using **sizeof** … but this migh
    than the sum of its parts

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec r;
    void *_r = &r;
    void *_e = &(r.eNum);
    void *_n = &(r.name);
    void *_s = &(r.salary);
    printf( "Size: %d\n" , sizeof( struct Rec ) );
    printf( "eNum offset: %d\n" , _e - _r );
    printf( "name offset: %d\n" , _n - _r );
    printf( "salary offset: %d\n" , _s - _r );
    return 0;
}
```

```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```



Structures (**structs**)

- When the compiler sees a **struct** … creates enough memory on the sta… store all of its contents
  - You can get the size of the memory a… a struct using **sizeof** … but this migh… than the sum of its parts

```c
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec r;
	void *_r = &r;
	void *_e = &(r.eNum);
	void *_n = &(r.name);
	void *_s = &(r.salary);
	printf( "Size: %d\n" , sizeof( struct Rec ) );
	printf( "eNum offset: %d\n" , _e - _r );
	printf( "name offset: %d\n" , _n - _r );
	printf( "salary offset: %d\n" , _s - _r );
	return 0;
}
```

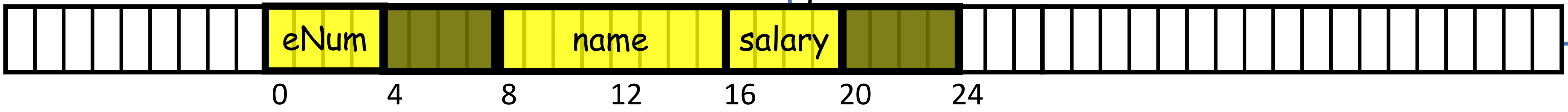
```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```

Structures (**structs**)

- When the compiler sees a **struct** … creates enough memory on the sta… store all of its contents
  - You can get the size of the memory a… a struct using **sizeof** … but this migh… than the sum of its parts
  - The members are laid out in order but there may be added padding!

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec r;
    void *_r = &r;
    void *_e = &(r.eNum);
    void *_n = &(r.name);
    void *_s = &(r.salary);
    printf( "Size: %d\n" , sizeof( struct Rec ) );
    printf( "eNum offset: %d\n" , _e - _r );
    printf( "name offset: %d\n" , _n - _r );
    printf( "salary offset: %d\n" , _s - _r );
    return 0;
}
```

```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```



Structures (**structs**)

- When the compiler sees a **struct** creates enough memory on the sta store all of its contents
  - You can get the size of the memory a a struct using **sizeof** … but this migh than the sum of its parts
  - The members are laid out in order but there may be added padding!
    1. Start members at offsets that are multiples of their alignment

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec r;
    void *_r = &r;
    void *_e = &(r.eNum);
    void *_n = &(r.name);
    void *_s = &(r.salary);
    printf( "Size: %d\n" , sizeof( struct Rec ) );
    printf( "eNum offset: %d\n" , _e - _r );
    printf( "name offset: %d\n" , _n - _r );
    printf( "salary offset: %d\n" , _s - _r );
    return 0;
}
```

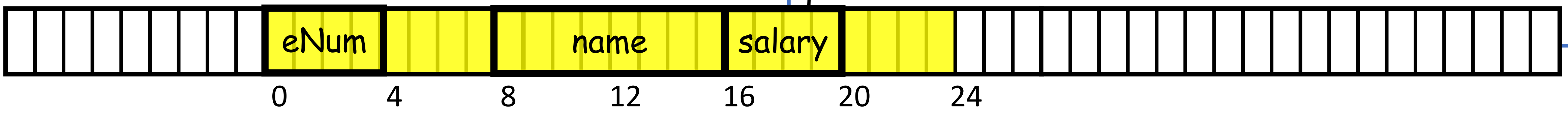
```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```

Structures (**structs**)

- When the compiler sees a **struct** … creates enough memory on the sta… store all of its contents
  - You can get the size of the memory a… a struct using **sizeof** … but this migh… than the sum of its parts
  - The members are laid out in order but there may be added padding!
    1. Start members at offsets that are multiples of their alignment
    2. Size should be a multiple of the size of the largest member

```c
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
int main( void )
{
    struct Rec r;
    void *_r = &r;
    void *_e = &(r.eNum);
    void *_n = &(r.name);
    void *_s = &(r.salary);
    printf( "Size: %d\n" , sizeof( struct Rec ) );
    printf( "eNum offset: %d\n" , _e - _r );
    printf( "name offset: %d\n" , _n - _r );
    printf( "salary offset: %d\n" , _s - _r );
    return 0;
}
```

```
>> ./a.out
Size: 24
eNum offset: 0
name offset: 8
salary offset: 16
>>
```

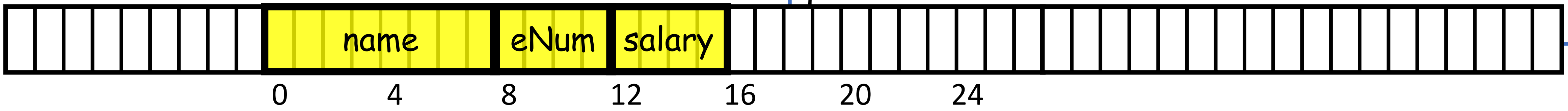


Structures (**structs**)

- When the compiler sees a **struct**
  creates enough memory on the sta
  store all of its contents
  - You can get the size of the memory a
    a struct using **sizeof** … but this migh
    than the sum of its parts
  - The members are laid out in order
    but there may be added padding!
    1. Start members at offsets that are multiples of their alignment
    2. Size should be a multiple of the size of the largest member

```
#include <stdio.h>
struct Rec
{
    const char * name;
    unsigned int eNum;
    float salary;
};
int main( void )
{
    struct Rec r;
    void *_r = &r;
    void *_n = &(r.name);
    void *_e = &(r.eNum);
    void *_s = &(r.salary);
    printf( "Size: %d\n" , sizeof( struct Rec ) );
    printf( "name offset: %d\n" , _n - _r );
    printf( "eNum offset: %d\n" , _e - _r );
    printf( "salary offset: %d\n" , _s - _r );
    return 0;
}
```

```
>> ./a.out
Size: 16
name offset: 0
eNum offset: 8
salary offset: 12
>>
```



Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
struct Rec Increase( struct Rec r , float s)
{
    r.salary += s;
    return r;
}
int main( void )
{
    struct Rec boss = { 1 , "misha" , 0.f };
    printf( "%g\t" , boss.salary );
    boss = Increase( boss , 1e6f );
    printf( "%g\n" , boss.salary );
    return 0;
}
```

```
>> ./a.out
0       1e+06
>>
```

Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions
  - On return, the entire **struct** (i.e. all its contents) is copied from the stack-frame of the called function to the stack-frame of the calling function

```
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
struct Rec Increase( struct Rec r , float s)
{
	r.salary += s;
	return r;
}
int main( void )
{
	struct Rec boss = { 1 , "misha" , 0.f };
	printf( "%g\t" , boss.salary );
	boss = Increase( boss , 1e6f );
	printf( "%g\n" , boss.salary );
	return 0;
}
```

```
>> ./a.out
0       1e+06
>>
```

Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions
  - Arguments are passed by value so the function sees a copy of the data in the **struct**

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
void Increase( struct Rec r , float s)
{
    r.salary += s;
}
int main( void )
{
    struct Rec boss = { 1 , "misha" , 0.f };
    printf( "%g\t" , boss.salary );
    Increase( boss , 1e6f );
    printf( "%g\n" , boss.salary );
    return 0;
}
```

```
>> ./a.out
0       0
>>
```

Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions
  - If you want to access the original data (or the **struct** is large and you don't want to duplicate it) you can pass a pointer
    - You can dereference the pointer and use the "." operator to access the member data

```
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
void Increase( struct Rec * r , float s)
{
    (*r).salary += s;
}
int main( void )
{
    struct Rec boss = { 1 , "misha" , 0.f };
    printf( "%g\t" , boss.salary );
    Increase( &boss , 1e6f );
    printf( "%g\n" , boss.salary );
    return 0;
}
```

```
>> ./a.out
0       1e+06
>>
```

Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions
  - If you want to access the original data (or the **struct** is large and you don't want to duplicate it) you can pass a pointer
    - You can dereference the pointer and use the "." operator to access the member data
    - Or you can use the "->" operator to access the member data directly from the pointer

```c
#include <stdio.h>
struct Rec
{
    unsigned int eNum;
    const char * name;
    float salary;
};
void Increase( struct Rec * r , float s)
{
    r->salary += s;
}
int main( void )
{
    struct Rec boss = { 1 , "misha" , 0.f };
    printf( "%g\t" , boss.salary );
    Increase( &boss , 1e6f );
    printf( "%g\n" , boss.salary );
    return 0;
}
```

```
>> ./a.out
0       1e+06
>>
```

Structures (**struct**s)

- Whole **struct**s can be assigned values and copied, and/or passed into or returned from functions
  - If a **struct** contains an array, the values are stored as part of the **struct**

  ⇒ If a function returns the **struct**, the values are copied to the calling function

  ⇒ Wrapping arrays within a **struct**, we can have functions that effectively return arrays.

```c
#include <stdio.h>
struct FourInts
{
	int ints[4];
};
struct FourInts Init( void )
{
	struct FourInts fourInts;
	for( int i=0 ; i<4 ; i++ ) fourInts.ints[i] = i;
	return fourInts;
}
int main( void )
{
	struct FourInts fi = Init();
	for( int i=0 ; i<4 ; i++ )
		printf( "%d] %d\n" , i , fi.ints[i] );
	return 0;
}
```

```
>> ./a.out
0] 0
1] 1
2] 2
3] 3
>>
```

Structures (**struct**s)

- You can nest **structs**
  - Since both "." and "->" associate left-to-right, the employee number of the lead is:

```
(mgmt.lead).eNum
(t->lead).eNum
mgmt.lead.eNum
t->lead.eNum
```

```
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
struct TeamRec
{
	struct Rec lead;
	struct Rec e1 , e2;
};
int main( void )
{
	…
	struct TeamRec mgmt;
	mgmt.lead = boss;
	mgmt.lead.salary *=2;
	TeamRec *t = &mgmt;
	…
}
```

Structures (**struct**s)

- You can nest **struct**s
- You can create arrays of **struct**s
  - Statically, on the stack

```
#include <stdio.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec staff[10];
	for( int i=0 ; i<10 ; i++ )
	{
		staff[i].eNum = i;
		…
	}
	return 0;
}
```

Structures (**struct**s)

- You can nest **structs**
- You can create arrays of **structs**
    - Statically, on the stack
    - Or dynamically on the heap

```
#include <stdio.h>
#include <stdlib.h>
struct Rec
{
	unsigned int eNum;
	const char * name;
	float salary;
};
int main( void )
{
	struct Rec *staff;
	staff = malloc( sizeof( struct Rec )*10 );
	for( int i=0 ; i<10 ; i++ )
	{
		staff[i].eNum = i;
		…
	}
	free( staff );
	return 0;
}
```

Structures (**struct**s)

- You can nest **struct**s
- You can create arrays of **struct**s
  - Statically, on the stack
  - Or dynamically on the heap
- You can declare a **struct** inside of a **struct**

```
#include <stdio.h>
#include <stdlib.h>
struct Pixel
{
	struct
	{
		unsigned char r , g , b;
	} color;
	struct
	{
		int x , y;
	} position;
};
int main( void )
{
	struct Pixel p;
	p.color.r = p.color.g = p.color.b = 255;
	p.position.x = p.position.y = 0;
	…
	return 0;
}
```

Structures (**struct**s)

- You can nest **struct**s
- You can create arrays of **struct**s
  - Statically, on the stack
  - Or dynamically on the heap
- You can declare a **struct** inside of a **struct**
  - Note that these lines are simultaneously:
    - Defining an (unnamed) struct with three unsigned chars, and
    - Declaring a member color of that type.

```
#include <stdio.h>
#include <stdlib.h>
struct Pixel
{
	struct
	{
		unsigned char r , g , b;
	} color;
	struct
	{
		int x , y;
	} position;
};
int main( void )
{
	struct Pixel p;
	p.color.r = p.color.g = p.color.b = 255;
	p.position.x = p.position.y = 0;
	…
	return 0;
}
```

Outline

- Lifetime and scope
- `struct`s
- `typedef`
- Random numbers
- Review questions

typedef

- Declaring / passing a **struct** requires adding the **struct** keyword

```
#include <stdio.h>
struct Rec
{
    unsigned int emplNum;
    const char * name;
    float salary;
};
void PrintRec( struct Rec r )
{
    printf( "Number: %d\n" , r.emplNum );
    printf( "Name: %s\n" , r.name );
    printf( "Salary: %.2f\n" , r.salary );
}
int main( void )
{
    struct Rec boss = { 1 , "misha" , 0.f };
    PrintRec( boss );
    return 0;
}
```

```
>> ./a.out
Number: 1
Name: misha
Salary: 0.00
>>
```

typedef

- Declaring / passing a **struct** requires adding the **struct** keyword
- We can use the **typedef** keyword to define a new "type" that has the keyword **struct** baked in:

  typdef <type> <alias>;

```c
#include <stdio.h>
struct _Rec
{
	unsigned int emplNum;
	const char * name;
	float salary;
};
typedef struct _Rec Rec;
void PrintRec( Rec r )
{
	printf( "Number: %d\n" , r.emplNum );
	printf( "Name: %s\n" , r.name );
	printf( "Salary: %.2f\n" , r.salary );
}
int main( void )
{
	Rec boss =  { 1 , "misha" , 0.f };
	PrintRec( boss );
	return 0;
}
```

typedef

- Declaring / passing a struct requires adding the struct keyword
- We can use the typedef keyword to define a new "type" that has the keyword struct baked in:

  typdef <type> <alias>;

- We can even apply it to the definition of the struct

```
#include <stdio.h>
typedef struct _Rec
{
    unsigned int emplNum;
    const char * name;
    float salary;
} Rec;

void PrintRec( Rec r )
{
    printf( "Number: %d\n" , r.emplNum );
    printf( "Name: %s\n" , r.name );
    printf( "Salary: %.2f\n" , r.salary );
}
int main( void )
{
    Rec boss = { 1 , "misha" , 0.f };
    PrintRec( boss );
    return 0;
}
```

typedef

- Declaring / passing a **struct** requires adding the **struct** keyword
- We can use the **typedef** keyword to define a new "type" that has the keyword **struct** baked in:

  typdef <type> <alias>;

- We can even apply it to the definition of the **struct**
- We can even omit the actual **struct** name altogether

```
#include <stdio.h>
typedef struct
{
    unsigned int emplNum;
    const char * name;
    float salary;
} Rec;

void PrintRec( Rec r )
{
    printf( "Number: %d\n" , r.emplNum );
    printf( "Name: %s\n" , r.name );
    printf( "Salary: %.2f\n" , r.salary );
}
int main( void )
{
    Rec boss = { 1 , "misha" , 0.f };
    PrintRec( boss );
    return 0;
}
```

Outline

- Lifetime and scope
- structs
- typedef
- Random numbers
- Review questions

Random numbers

`stdlib.h` declares two functions for generating random numbers

```
int rand( void );
```

- Returns a random integer value between 0 and `RAND_MAX`
- `RAND_MAX` is a constant (at least 32767)
- Each call to `rand` creates a new random number

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d < %d\n" , rand() , RAND_MAX );
    printf( "%d < %d\n" , rand() , RAND_MAX );
    return 0;
}
```

Random numbers

`stdlib.h` declares two functions for generating random numbers

int rand(void);

- Returns a random integer value between 0 and RAND_MAX
- RAND_MAX is a constant (at least 32767)
- Each call to rand creates a new random number

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d < %d\n" , rand() , RAND_MAX );
    printf( "%d < %d\n" ,
    return 0;
}
```

```
>> ./a.out
1804289383 < 2147483647
846930886 < 2147483647
>>
```

Random numbers

`stdlib.h` declares two functions for generating random numbers

```
void srand( unsigned int );
```

- Seeds the random number generator
- Calling `rand` after the random number has been seeded will consistently generate the same set of random numbers.
- Useful for debugging (for consistency)
- Useful for trying different values

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    srand( 1 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 2 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 1 );
    printf( "%d , %d\n" , rand() , rand() );
    return 0;
}
```

Random numbers

`stdlib.h` declares two functions for generating random numbers

void srand(unsigned int);

- Seeds the random number generator
- Calling rand after the random number has been seeded will consistently generate the same set of random numbers.
- Useful for debugging (for consistency)
- Useful for trying different values

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    srand( 1 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 2 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 1 );
    printf( "%d , %
    return 0;
}
```

```
>> ./a.out
846930886 , 1804289383
1738766719 , 1505335290
846930886 , 1804289383
>>
```

Warning:
Seeding the random number with a value of 1 has a special meaning. It re-seeds the random number generator to its original state.

On some machines the “original state” is being seeded with 0, so that `srand(0)` and `srand(1)` do the same thing.

- Seeds the random number generator
- Calling `rand` after the random number has been seeded will consistently generate the same set of random numbers.
- Useful for debugging (for consistency)
- Useful for trying different values

```
int main( void )
{
    srand( 1 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 2 );
    printf( "%d , %d\n" , rand() , rand() );
    srand( 1 );
    printf( "%d , %
    return 0;
}
```

```
>> ./a.out
846930886 , 1804289383
1738766719 , 1505335290
846930886 , 1804289383
>>
```

Random numbers

We can use **rand** to generate random numbers in an integer range

```
#include <stdio.h>
#include <stdlib.h>

int myRand( int low , int high )
{
	return low + rand() % ( high - low );
}
int main( void )
{
	printf( "%d , %d\n" , myRand(2,6) , myRand(2,6) );
	printf( "%d , %d\n" , myRand(16,26) , myRand(16,26) );
	return 0;
}
```

```
>> ./a.out
3, 5
21, 23
>>
```

Random numbers

We

Note:
This will create random numbers in the range [low,high).

```
#include <stdio.h>
#include <stdlib.h>

int myRand( int low , int high )
{
    return low + rand() % ( high - low );
}
int main( void )
{
    printf( "%d , %d\n" , myRand(2,6) , myRand(2,6) );
    printf( "%d , %d\n" , myRand(16,26) , myRand(16,26) );
    return 0;
}
```

```
>> ./a.out
3, 5
21, 23
>>
```

Random numbers

We can use **rand** to generate random numbers in a floating point range

```
#include <stdio.h>
#include <stdlib.h>

float myRand( float low , float high )
{
    return low + (float)rand() / RAND_MAX * ( high - low );
}
int main( void )
{
    printf( "%f , %f\n" , myRand(2,6) , myRand(2,6) );
    printf( "%f , %f\n" , myRand(16,26) , myRand(16,26) );
    return 0;
}
```

```
>> ./a.out
3.577532 , 5.360751
23.984400 , 23.830992
>>
```

Outline

- Lifetime and scope
- `struct`s
- typedef
- Random numbers
- Review questions

Review questions

1. What is a `struct` in c?

Review questions

2. How are the fields of a `struct` passed into a function – by value or by reference?

Review questions

3. What is the size of a `struct`? What is structure padding in C?

Review questions

4. What is the difference between lifetime and scope of a variable?

Review questions

5. What is variable shadowing (i.e. hiding)?

Exercise 5-1

- Website -> Course Materials -> Ex5-1