Intermediate Programming

Day 13

Outline

- Exercise 12
- Lifetime and scope
- structs
- typedef
- Review questions

Exercise 12

Declare and define search.

bsearch.c

```
…
int *search( int *start , int *end , int s_val );

int main( void ) {
    …
}

int *search( int *start , int *end , int s_val )
{
    if( start==end ) return NULL;
    int *mid = start + (end-start)/2;
    if( *mid==s_val ) return mid;
    else if( *mid<s_val ) return search( mid+1 , end , s_val );
    else return search( start , mid , s_val );
}
```

Exercise 12

Compute the index of the matching element

bsearch.c

```
...
int *search( int *start , int *end , int s_val );

int main( void ) {
        ...
        int arr1[] = { 11, 119, 318, 518, 573, 750, 757, 809, 813, 994 };

        // example of a successful search
        pos = search(arr1, arr1 + 10, 809);
        assert(pos != NULL);
        assert(*pos == 809);
        index = pos - arr1;
        assert(7 == index);
        ...
}

int *search( int *start , int *end , int s_val )
{
        if( start==end ) return NULL;
        int *mid = start + (end-start)/2;
        if( *mid==s_val ) return mid;
        else if( *mid<s_val ) return search( mid+1 , end , s_val );
        else return search( start , mid , s_val );
}
```

Exercise 12

Declare the unit array.

sudokuHelper.c

```
...
int *makeCol( int *table ) {
        // TODO: declare the unit variable (array of 9 integers, to be returned)
        int *unit = malloc( sizeof(int) * SIZE );
        if( !unit )
        {
                fprintf( stderr , "[ERROR] Failed to allocate unit\n" );
                return NULL;
        }
        ...
}

int *makeCube( int *table ) {
        // TODO: declare the unit variable (array of 9 integers, to be returned)
        int *unit = malloc( sizeof(int) * SIZE );
        if( !unit )
        {
                fprintf( stderr , "[ERROR] Failed to allocate unit\n" );
                return NULL;
        }
        ...
}
...
```

Exercise 12

Call check on current row and add to variable good

sudokuHelper.c

```
...
int checkRows( int table[][SIZE] ) {
        int good = 0;
        for (int r = 0; r < SIZE; r++) {
                // TODO: call check on current row and add to variable good
                good += check( table[r] );
        }
        return ( good==SIZE);
}
...
```

Exercise 12

Call **makeCol/makeCube** on current column/cube and assign result to variable **column/cube**

sudokuHelper.c

```
...
int checkCols( int table[][SIZE] ) {
        int good = 0;
        int *column;
        for( int c=0 ; c<SIZE ; c++ ) {
                // TODO: call makeCol on current column and assign result to variable column
                column = makeCol( table[0]+c );
                good += check(column);
        }
        return ( good==SIZE );
}

int checkCubes(int table[][SIZE]) {
        int good = 0;
        int *cube;
        for( int r=0 ; r<SIZE ; r+=3)
                for( int c=0 ; c<SIZE ; c+=3 ) {
                        // TODO: call makeCube on current cube and assign result to variable cube
                        cube = makeCube( table[r]+c );
                        good += check(cube);
                }
        return (good == SIZE);
}
...
```

Exercise 12

Find and fix the memory leaks

```
>> valgrind --leak-check=full --show-leak-kinds=all ./main puzzle1.txt
...
==3710153== HEAP SUMMARY:
==3710153==     in use at exit: 1,120 bytes in 19 blocks
==3710153==   total heap usage: 21 allocs, 2 frees, 10,336 bytes allocated
==3710153==
==3710153== 324 bytes in 9 blocks are definitely lost in loss record 1 of 3
==3710153==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==3710153==    by 0x4013F9: makeCol (sudokuHelpers.c:22)
==3710153==    by 0x4015A7: checkCols (sudokuHelpers.c:85)
==3710153==    by 0x401317: main (in /home/misha/CS220/exercises/ex12/main)
```

Exercise 12

Find and fix the memory leaks

sudokuHelper.c

```
…
21.   int* makeCol(int *table) {
22.         int *unit = malloc( sizeof(int) * SIZE );
23.         if( !unit )
24.         {
25.               fprintf( stderr , "[ERROR] Failed to allocate unit\n" );
26.               return NULL;
27.         }
 …
81.   int checkCols( int table[][SIZE] ) {
82.         int good = 0;
83.         int * column;
84.         for( int c=0 ; c<SIZE ; c++ ) {
85.               column = makeCol( table[0]+c );
86.               good += check(column);
87.         }
88.         return ( good==SIZE );
89.   }
…
```

```
>> valgrind --leak-check=full --show-leak-kinds=all ./main puzzle1.txt
...
==3710153== HEAP SUMMARY:
==3710153==     in use at exit: 1,120 bytes in 19 blocks
==3710153==   total heap usage: 21 allocs, 2 frees, 10,336 bytes allocated
==3710153==
==3710153== 324 bytes in 9 blocks are definitely lost in loss record 1 of 3
==3710153==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==3710153==    by 0x4013F9: makeCol (sudokuHelpers.c:22)
==3710153==    by 0x4015A7: checkCols (sudokuHelpers.c:85)
==3710153==    by 0x401317: main (in /home/misha/CS220/exercises/ex12/main)
```

Exercise 12

Find and fix the memory leaks

sudokuHelper.c

```
…
int* makeCol(int *table) {
    int *unit = malloc( sizeof(int) * SIZE );
    if( !unit )
    {
        fprintf( stderr , "[ERROR] Failed to allocate unit\n" );
        return NULL;
    }
 …
int checkCols( int table[][SIZE] ) {
    int good = 0;
    int * column;
    for( int c=0 ; c<SIZE ; c++ ) {
        column = makeCol( table[0]+c );
        good += check(column);
        free( column );
    }
    return ( good==SIZE );
}
int checkCubes(int table[][SIZE]) {
    int good = 0;
    int * cube;
    for (int r = 0; r < SIZE; r += 3)
        for (int c = 0; c < SIZE; c += 3) {
            cube = makeCube( table[r]+c );
            good += check(cube);
            free( cube );
        }
    return ( good==SIZE );
}
…
```

Exercise 12

Find and fix the memory leaks

sudokuHelper.c

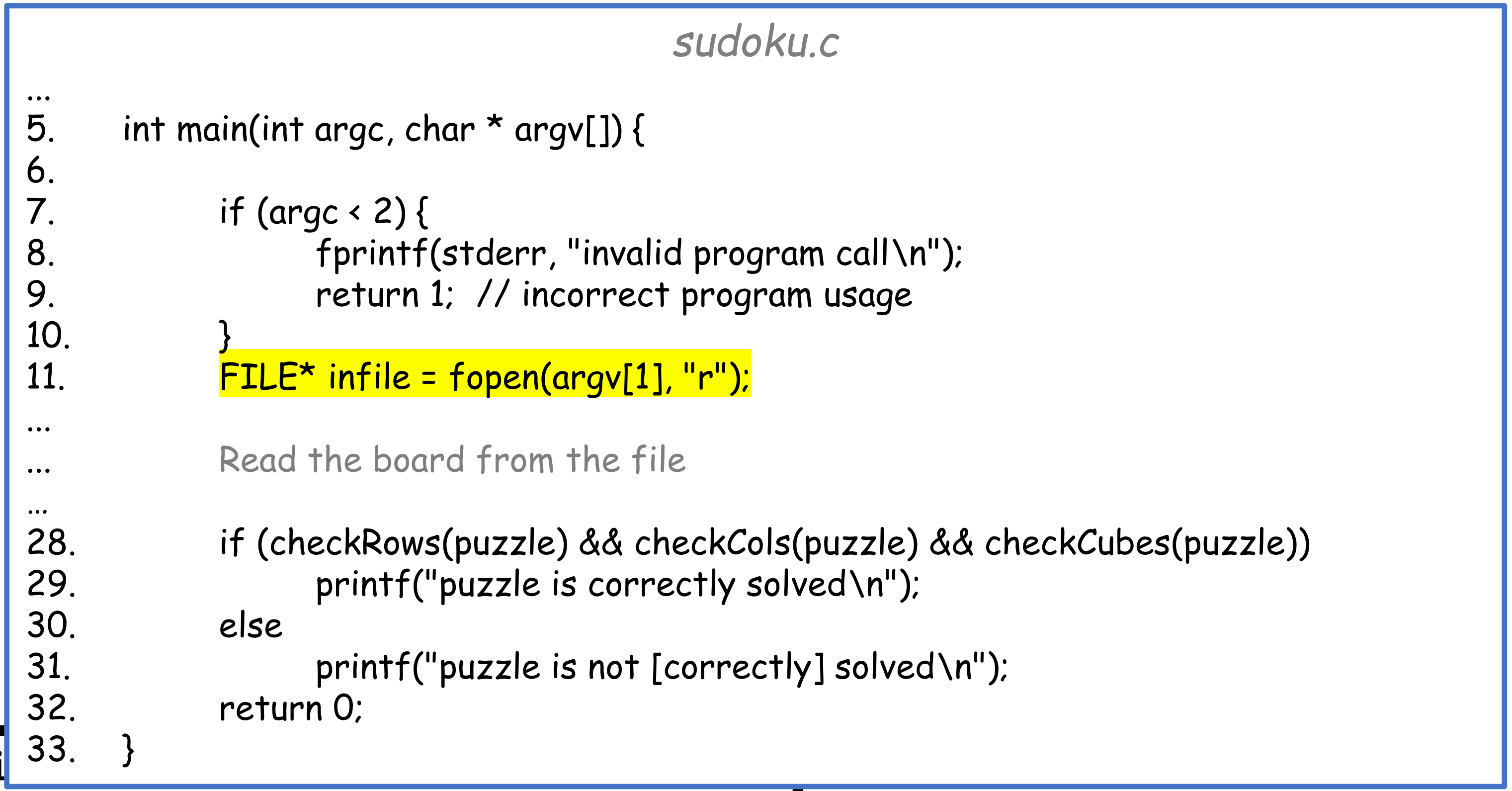
```
…
int* makeCol(int *table) {
        int *unit = malloc( sizeof(int) * SIZE );
        if( !unit )
        {
                fprintf( stderr , "[ERROR] Failed to allocate unit\n" );
                return NULL;
        }
 …
int checkCols( int table[][SIZE] ) {
        int good = 0;
        int * column;
        for( int c=0 ; c<SIZE ; c++ ) {
                column = makeCol( table[0]+c );
                good += check(column);
                free( column );
```

```
{
r]+c );
```

```
>> valgrind --leak-check=full --show-leak-kinds=all ./main puzzle1.txt
...
==3923831== HEAP SUMMARY:
==3923831==     in use at exit: 472 bytes in 1 blocks
==3923831==   total heap usage: 21 allocs, 20 frees, 10,336 bytes allocated
==3923831==
==3923831== 472 bytes in 1 blocks are still reachable in loss record 1 of 1
==3923831==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==3923831==    by 0x48FA46E: __fopen_internal (iofopen.c:65)
==3923831==    by 0x4011E9: main (sudoku.c:11)
==3923831==
==3923831== LEAK SUMMARY:
...
```

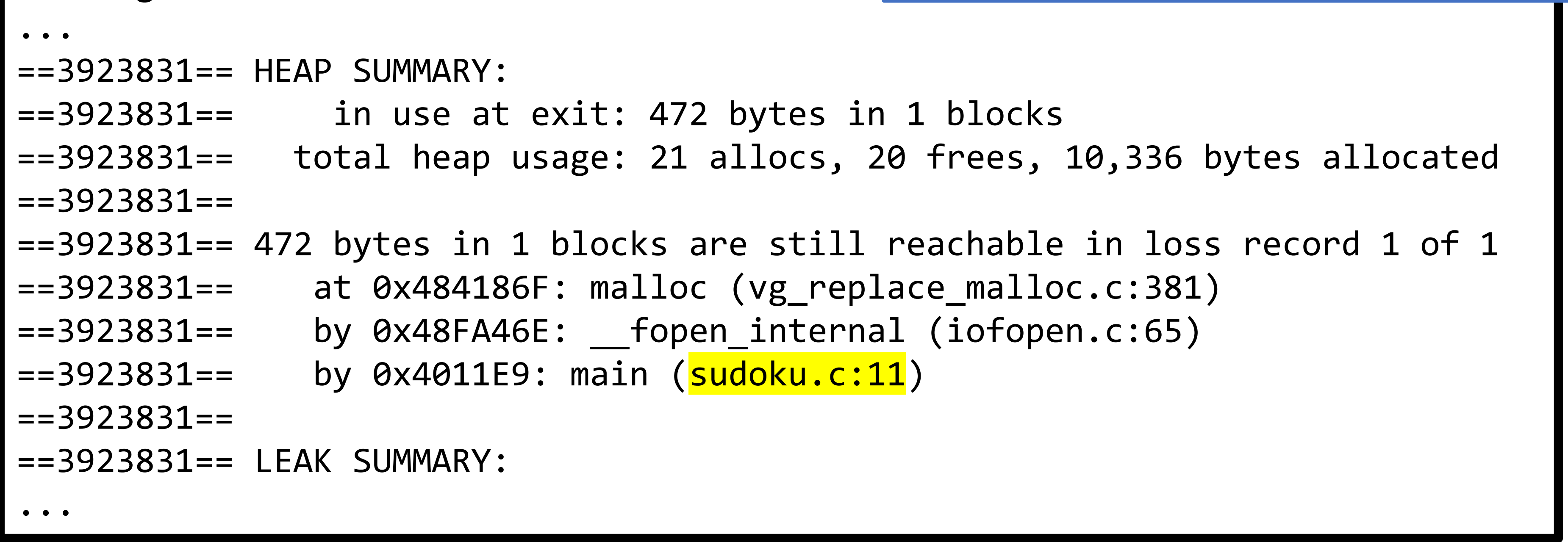
Exercise 12

Find and fix the memory leaks



sudoku.c

```
...
5.    int main(int argc, char * argv[]) {
6.
7.        if (argc < 2) {
8.            fprintf(stderr, "invalid program call\n");
9.            return 1;  // incorrect program usage
10.       }
11.       FILE* infile = fopen(argv[1], "r");
...
...       Read the board from the file
...
28.       if (checkRows(puzzle) && checkCols(puzzle) && checkCubes(puzzle))
29.           printf("puzzle is correctly solved\n");
30.       else
31.           printf("puzzle is not [correctly] solved\n");
32.       return 0;
33.   }
```



```
>> valgrind --leak-check=full --show-leak-ki
...
==3923831== HEAP SUMMARY:
==3923831==     in use at exit: 472 bytes in 1 blocks
==3923831==   total heap usage: 21 allocs, 20 frees, 10,336 bytes allocated
==3923831==
==3923831== 472 bytes in 1 blocks are still reachable in loss record 1 of 1
==3923831==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==3923831==    by 0x48FA46E: __fopen_internal (iofopen.c:65)
==3923831==    by 0x4011E9: main (sudoku.c:11)
==3923831==
==3923831== LEAK SUMMARY:
...
```

Exercise 12

Find and fix the memory leaks

sudoku.c

```
...
int main(int argc, char * argv[]) {

    if (argc < 2) {
        fprintf(stderr, "invalid program call\n");
        return 1;  // incorrect program usage
    }
    FILE* infile = fopen(argv[1], "r");
...
...     Read the board from the file
...
    if (checkRows(puzzle) && checkCols(puzzle) && checkCubes(puzzle))
        printf("puzzle is correctly solved\n");
    else
        printf("puzzle is not [correctly] solved\n");
    fclose( infile );
    return 0;
}
```

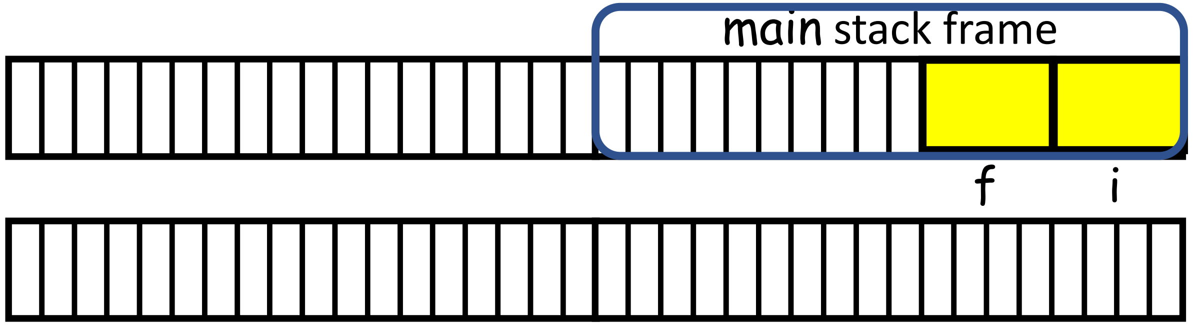
```
>> valgrind --leak-check=full --show-leak-ki
...
==3720658== HEAP SUMMARY:
==3720658==     in use at exit: 0 bytes in 0 blocks
==3720658==   total heap usage: 21 allocs, 21 frees, 10,336 bytes allocated
==3720658==
==3720658== All heap blocks were freed -- no leaks are possible
```

Outline

- Exercise 12
- Lifetime and scope
- structs
- typedef
- 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

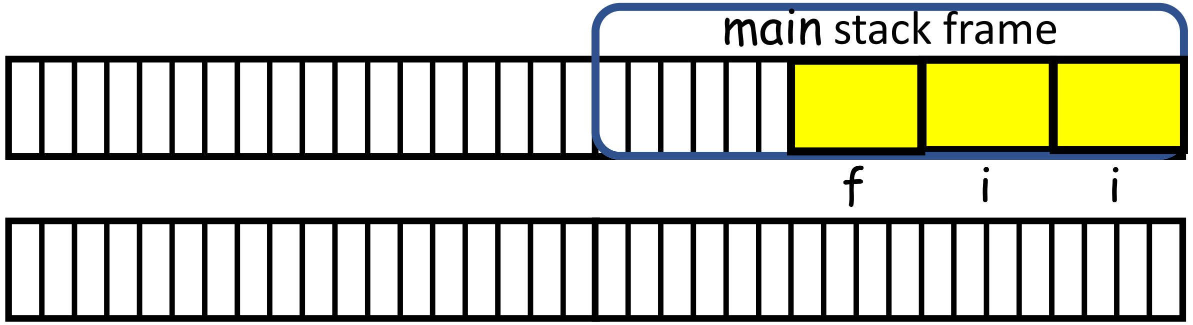
Q: What are the lifetimes of the variables i?

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 and are declared at the beginning)

```
#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 and are declared at the beginning)
  - 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

```
#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 and are declared at the beginning)
  - 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
    - 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 and are declared at the beginning)
  - 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

```
#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

- Variabl
  - In ge … n's duration (ass hidd the same name and are declared at the beginning)
  - 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

<u>Note</u>:
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*.)

```
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

- 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
>>
```

Note:

Like `static` variables, global variables do not reside on the stack.

(They too are stored in the *data segment*.)

Variable lifetime and scope

```
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

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.

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).

```
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

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

Variable lifetime and scope

```
int count = 3;
```

foo.c

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).

```
#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

✓ Boolean global variables could be useful for debugging if you only want to `printf` within one function based on a condition being met in a different function.

Outline

- Exercise 12
- Lifetime and scope
- structs
- typedef
- 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 (structs)

- 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 (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

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

struct Rec boss;
struct Rec assistant;
```

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
  - Can initialize members using array syntax
    - Variable order must match declaration order

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

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

struct Rec boss;
struct Rec assistant;
```

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
  - Can initialize members using array syntax
  - Or member-by-member, using the "." operator

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

struct Rec boss;
struct Rec assistant;
```

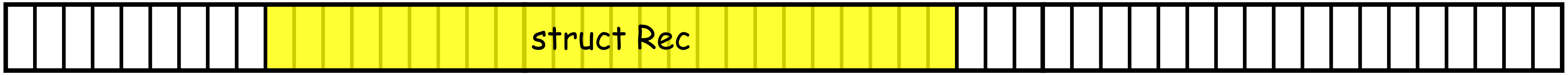
```
boss = { 1 , "misha" , 0.f };
```

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

Structures (**struct**s)

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

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

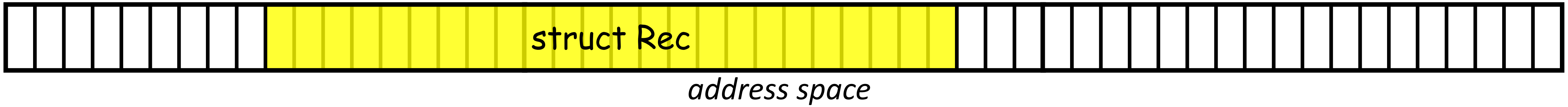


address space

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** …

```
#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;
}
```

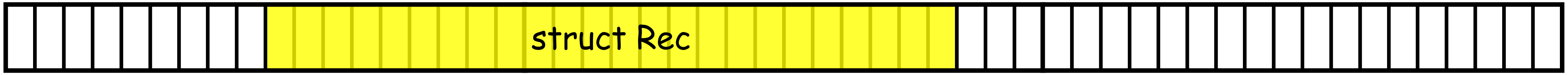


Structures (**struct**s)

- 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** … but this might be larger than the sum of its parts

```
#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
>>
```



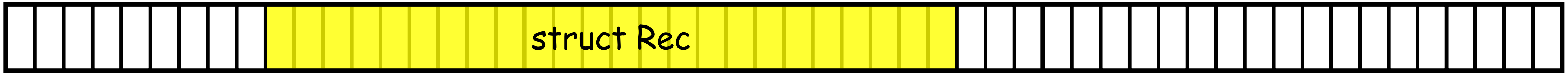
address space

Structures (**struct**s)

- 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

```
#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
>>
```



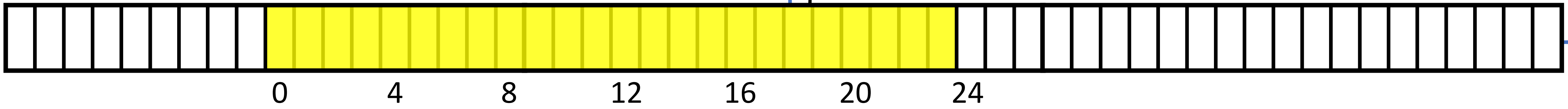
address space

Structures (**struct**s)

- 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
>>
```

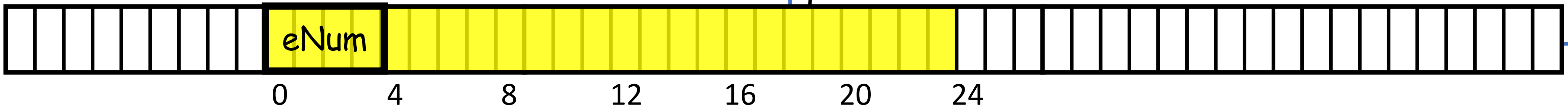


Structures (**structs**)

- 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
>>
```

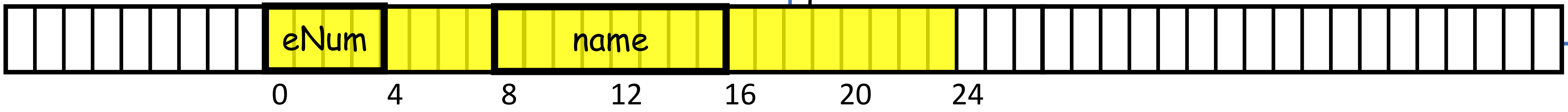


Structures (**structs**)

- 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
>>
```

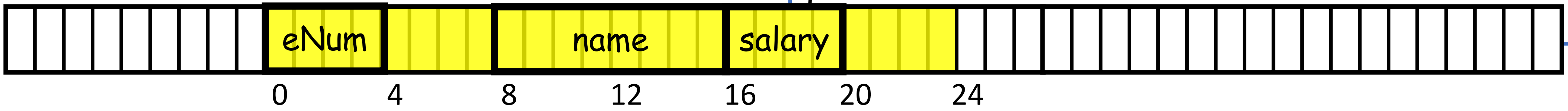


Structures (**structs**)

- 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

```
#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
>>
```

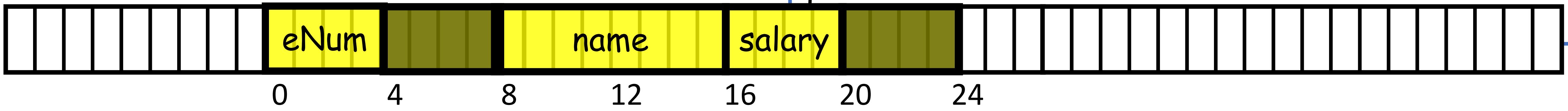


Structures (**struct**s)

- 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
>>
```

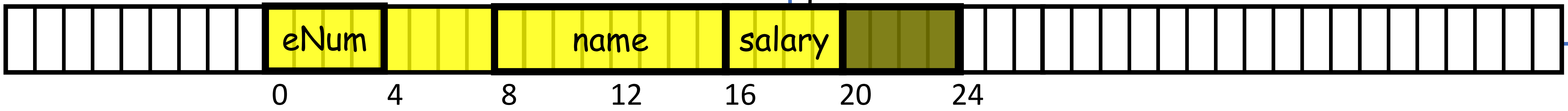


Structures (**struct**s)

- 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
>>
```

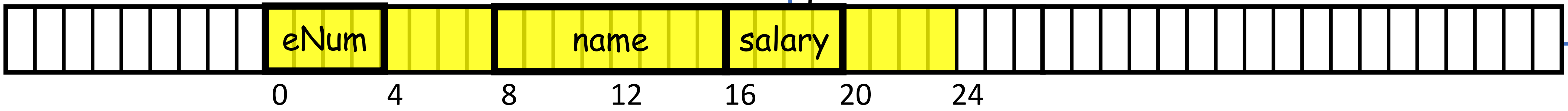


Structures (**structs**)

- 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

```
#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
>>
```

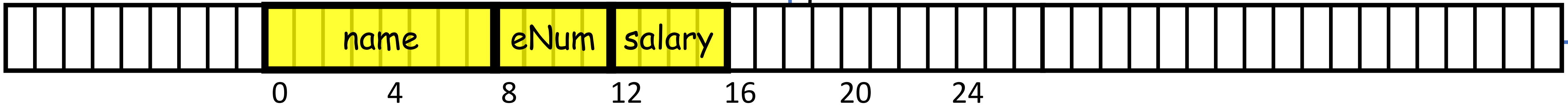


Structures (**struct**s)

- 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)

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

```c
#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)

- **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)

- structs 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)

- **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 (structs)

- structs 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

```
#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)

- **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.

```
#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 **struct**s
  - 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 **struct**s
- You can create arrays of **struct**s
  - 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 simultaneously:
    - Define an (unnamed) **struct** with three **unsigned char**s, and
    - Declare 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

- Exercise 12
- Lifetime and scope
- structs
- typedef
- 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>;

```
#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;
}
```

*This is OK unless we need to know the struct's name within the struct.

Outline

- Exercise 12
- Lifetime and scope
- `struct`s
- typedef
- Review questions

Review questions

1. What is a `struct` in c?

A user defined type which is a collection of variables (often heterogeneously-typed) that are bundled together as a unit under a single name

Review questions

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

By value

Review questions

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

The size of a struct is at least the number of bytes needed to store the data. It may be padded either to align the members or to ensure that the total size is a multiple of the largest member’s size.

Review questions

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

Lifetime describes how long the variable resides in memory.

Scope describes when it is accessible.

Review questions

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

When a variable goes out of scope because another variable with the same name is brought into scope.

Review questions

6. What is the output of this program?

```
0; 3; 5; 2;
```

(Recall that global variables are initialized to zero.)

```
#include <stdio.h>
int foo;
void bar( void )
{
        int foo = 3;
        {
                extern int foo;
                printf( "%d; “ , foo );
                foo = 2;
        }
        printf( "%d; ", foo );
}
void baz( void ) { printf( "%d; “ , foo ); }
int main( void )
{
        {
                int foo = 5;
                bar();
                printf( "%d; “ , foo );
        }
        baz();
        return 0;
}
```

Exercise 13

- Website -> Course Materials -> Exercise 13