CIS 190: C/C++ Programming

Lecture 2

Not So Basics

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

What is Separate Compilation?

Why Use Separate Compilation?

- organize code into collections of smaller files that can be compiled individually

- can separate based on:
  - a user-made “library” (e.g., math functions)
  - related tasks (e.g., functions for handling a data structure)
  - sub-parts of the program (e.g., reading user input)

Example: Homework 2 Files

```
void PrintTrain(...);
void AddTrainCar(...);

int main()
{
   [...]
}

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

hw2.c

Example: Homework 2 Files

```
void PrintTrain(...);
void AddTrainCar(...);

int main()
{
   [...]
}

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

hw2.c

trains.h

trains.c

Example: Homework 2 Files

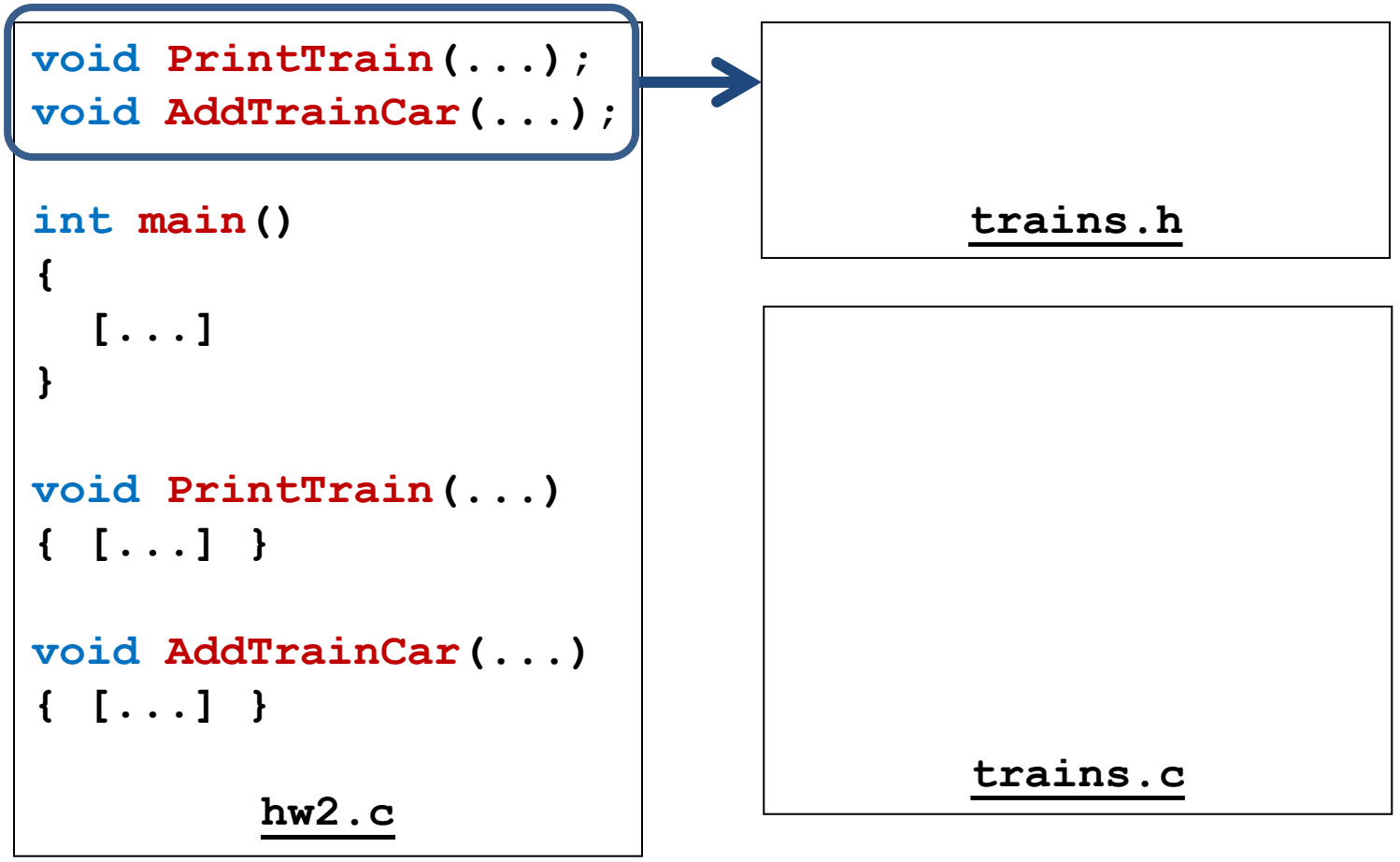
```
void PrintTrain(...);
void AddTrainCar(...);

int main()
{
   [...]
}

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

hw2.c

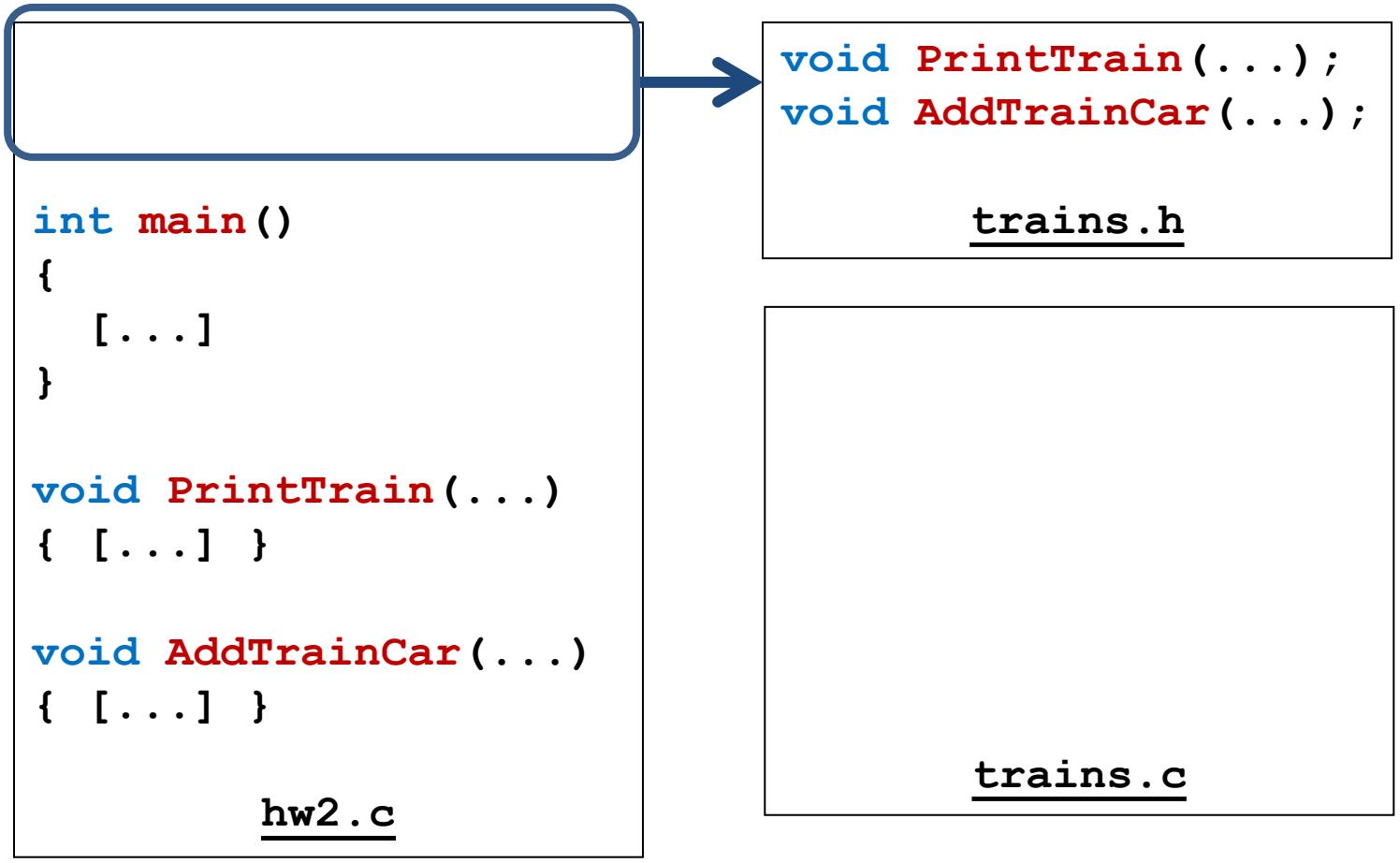
trains.h

trains.c

Example: Homework 2 Files



Example: Homework 2 Files



Example: Homework 2 Files

```
int main()
{
   [...]
}

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

trains.c

Example: Homework 2 Files

```
int main()
{
   [...]
}

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

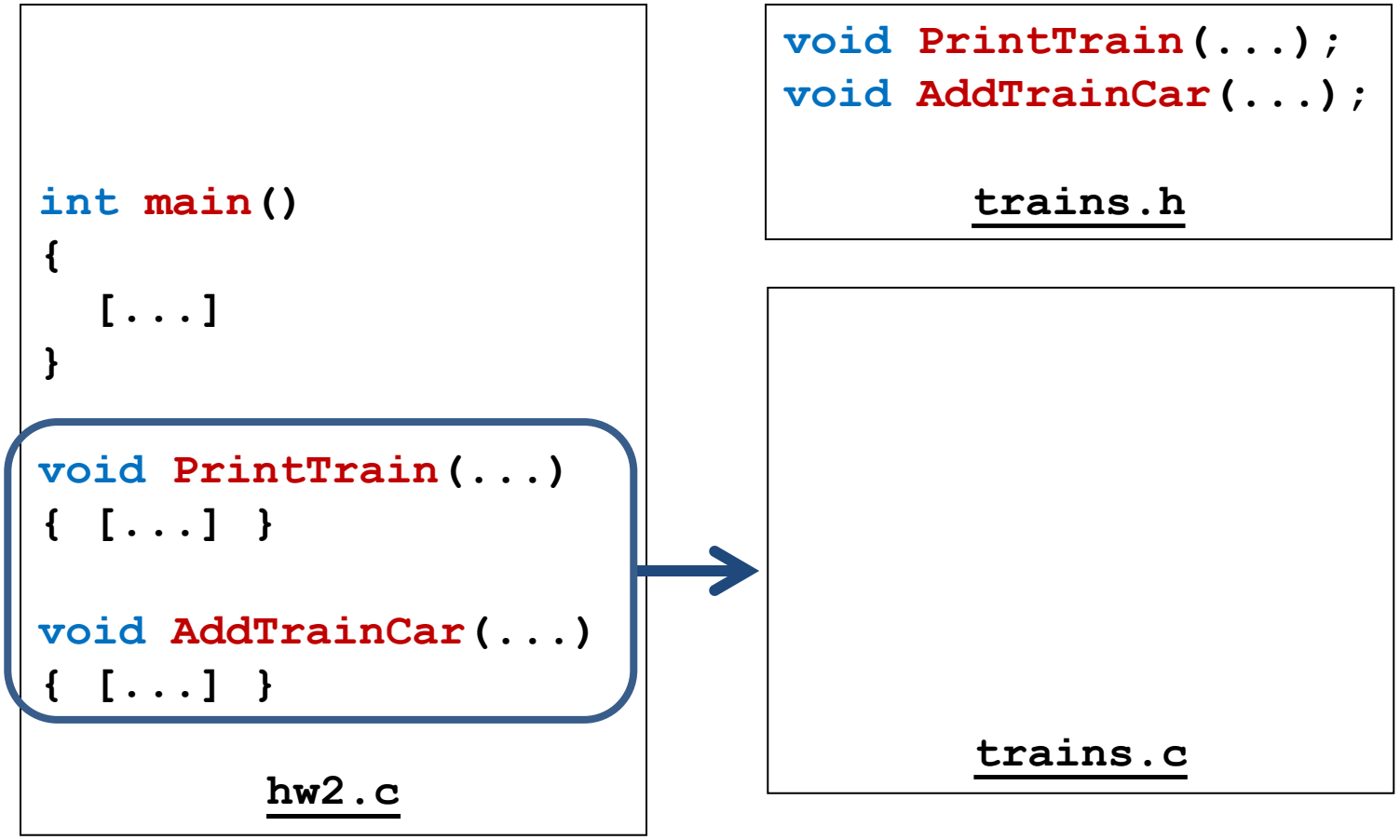
hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

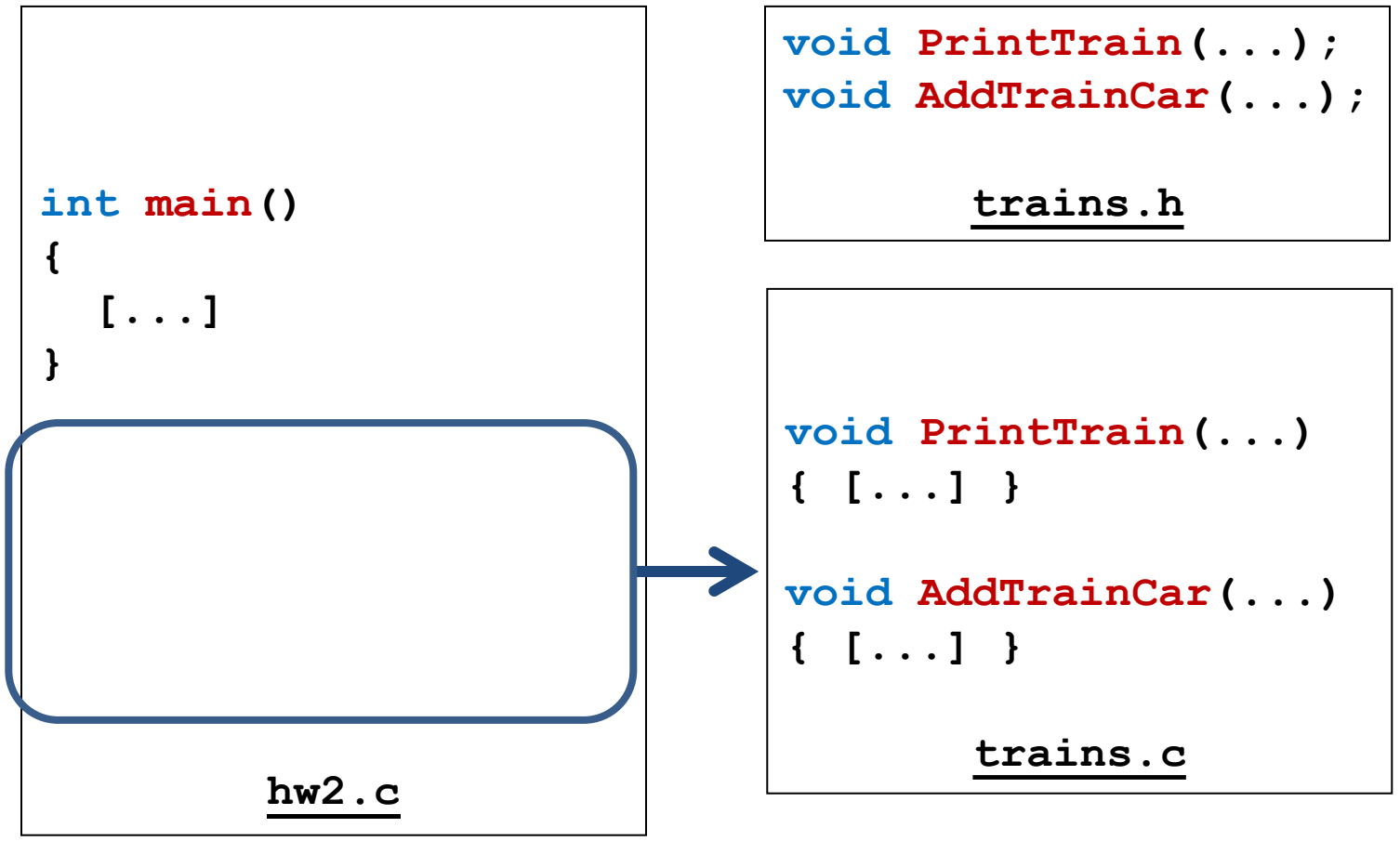
trains.h

trains.c

Example: Homework 2 Files



Example: Homework 2 Files



Example: Homework 2 Files

```
int main()
{
   [...]
}
```

hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

trains.c

Example: Homework 2 Files

```
int main()
{
   [...]
}
```

hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

trains.c

Example: Homework 2 Files

```
int main()
{
   [...]
}
```

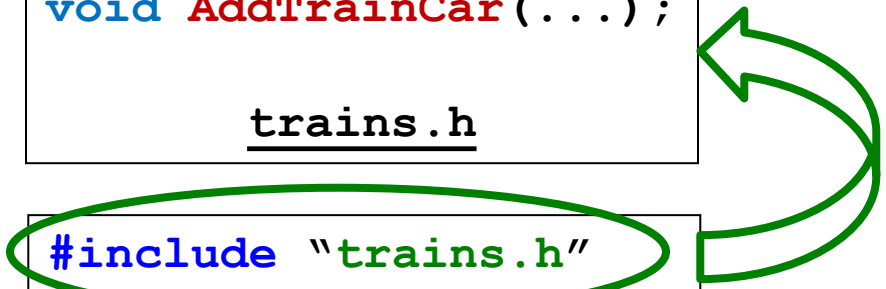
hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

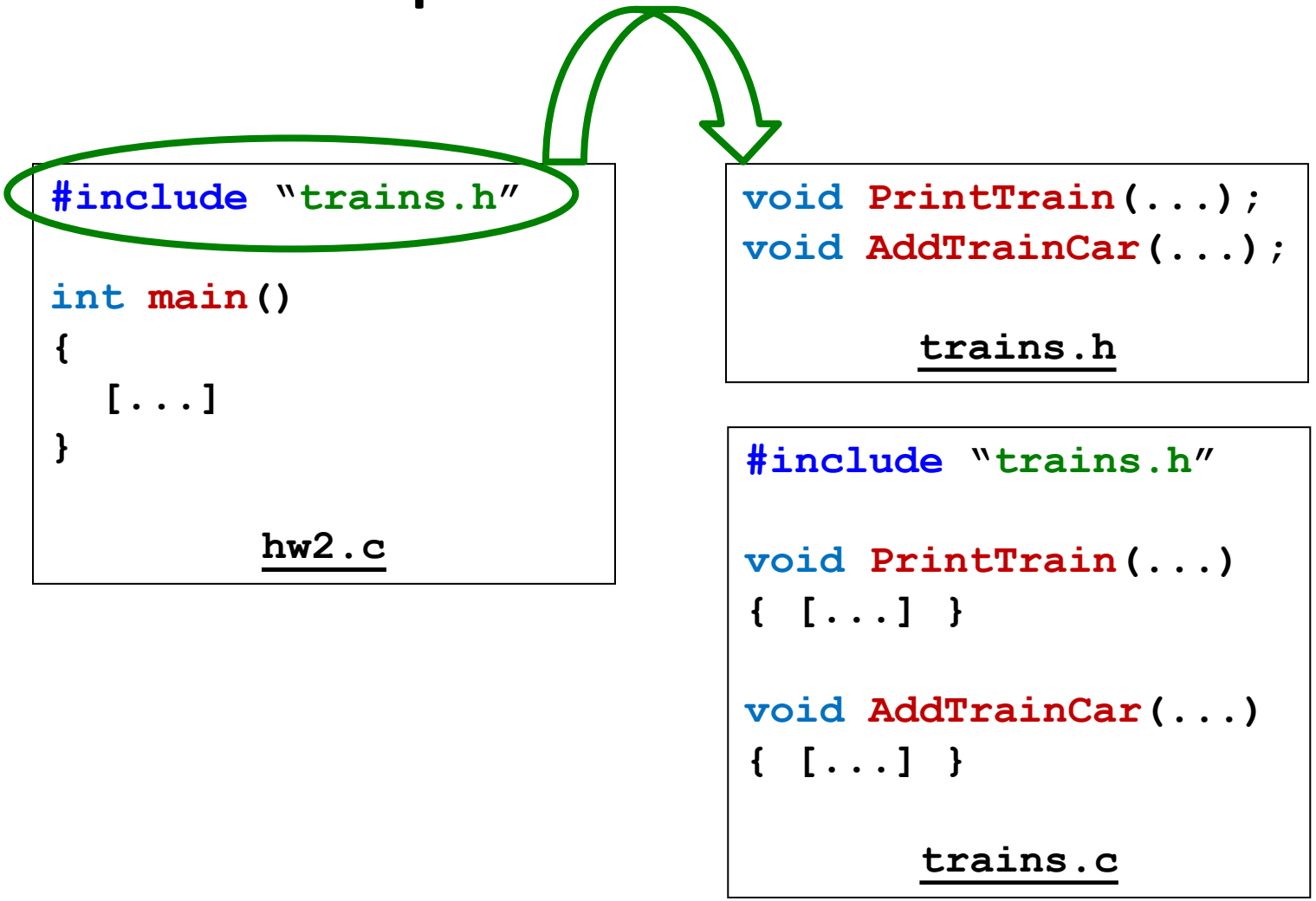
trains.h

```
#include "trains.h"

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

trains.c



Example: Homework 2 Files



Example: Homework 2 Files

```
#include "trains.h"

int main()
{
   [...]
}
```

hw2.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
#include "trains.h"

void PrintTrain(...)
{ [...] }

void AddTrainCar(...)
{ [...] }
```

trains.c

Separate Compilation

- need to **`#include "fileName.h"`** at top of any **`.c`** file using the functions prototypes inside that **`.h`** file

- for local files we use quotes

  `"filename.h"`

- for libraries we use carats

  `<stdio.h>`

Separate Compilation

- after a program is broken into multiple files, the individual files must be:
  - compiled separately
    - using `gcc` and the `-c` flag
  - linked together
    - using `gcc` and the created `.o` (object) files

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
> gcc -c -Wall hw2.c
```

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

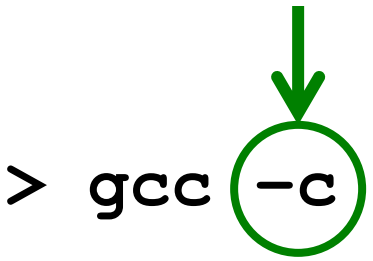
trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

tells the compiler we're compiling separately

- stops before linking
- won't throw an error if everything's not available



Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
> gcc -c -Wall hw2.c
```

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
> gcc -c -Wall hw2.c
```

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
> gcc -c -Wall hw2.c
```

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
```

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
```

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
***OBJECT FILE***
```

trains.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
```

Compiling Multiple .c Files

```
#include "trains.h"

int main()
{ [...] }
```

hw2.c

```
***OBJECT FILE***
```

hw2.o

```
***OBJECT FILE***
```

trains.o

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
```

Linking Multiple .o Files

```
#include "trains.h"

int main()
{ [...] }
```
hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```
trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```
trains.h

```
***OBJECT FILE***
```
hw2.o

```
***OBJECT FILE***
```
trains.o

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
> gcc -Wall hw2.o
  trains.o
```

Linking Multiple .o Files

```
#include "trains.h"

int main()
{ [...] }
```

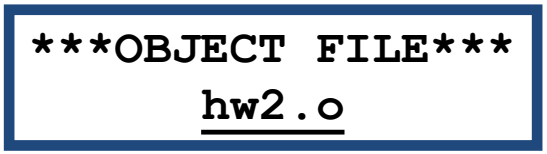
hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h




```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
> gcc -Wall hw2.o
  trains.o
```

Linking Multiple .o Files

```
#include "trains.h"

int main()
{ [...] }
```

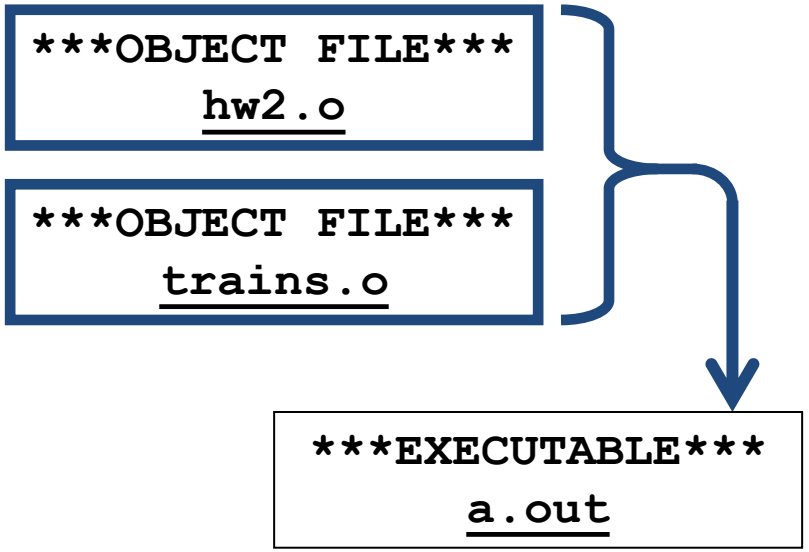
hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```

trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```

trains.h



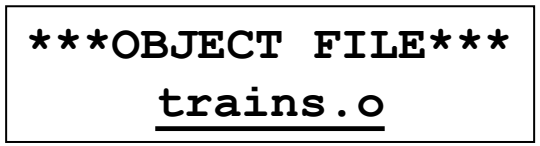

```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
> gcc -Wall hw2.o
  trains.o
```

Linking Multiple .o Files

```
#include "trains.h"

int main()
{ [...] }
```
hw2.c

```
#include "trains.h"

void PrintTrain(...)
{ [...] }
void AddTrainCar(...)
{ [...] }
```
trains.c

```
void PrintTrain(...);
void AddTrainCar(...);
```
trains.h




```
> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
> gcc -Wall hw2.o
  trains.o
```

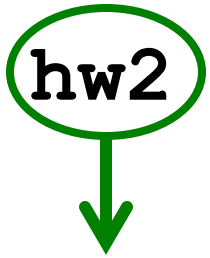
Naming Executables

- if you'd prefer to name the executable something other than **`a.out`**, use the **`-o`** flag

```
> gcc –Wall hw2.o trains.o
```

becomes

```
> gcc –Wall hw2.o trains.o –o hw2
```



name of the executable

- and to run it, you just type

```
> ./hw2
```

Common Mistakes

- **Do not**:
  - use **`#include`** for .c files

    ```
    #include "trains.c" – NO!
    ```

  - use **`#include`** *inside* a .h file

- **Do** be conservative:
  - only **`#include`** those files whose function prototypes are needed

Common Error Message

- if you receive this error:

"**`undefined reference to 'fxnName'`**"

- the linker can't find a function called fxnName

- 99% of the time, this is because fxnName was spelled wrong
  - could be in the definition/prototype or one of the times the function is called

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Structures

- collection of variables under one name
  - member variables can be of different types

- use structures (or ***structs***)
  - to keep related data together
  - to pass fewer arguments

An Example

- an example structure that represents a CIS class, which has the following ***member variables***:
  - an integer variable for the class number
  - string variables for the room and class title

```
struct cisClass
{
   int  classNum;
   char room [20];
   char title [30];
} ;
```

Example Structures

- point in 3-dimensional space
- mailing address
- student information

Example Structures

- for reference:

```
struct structName
{
   varType1 varName1;
   varType2 varName2;
   ...
   varTypeN varNameN;
} ;
```

Using Structs

- to declare a variable of type struct cisClass:

```
struct cisClass cis190;
```

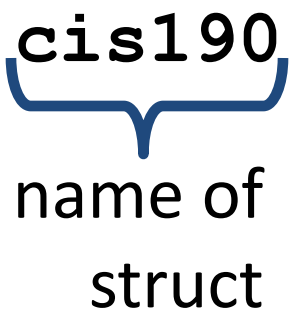
- to access a struct's members, use ***dot notation***:

Using Structs

- to declare a variable of type struct cisClass:

```
struct cisClass cis190;
```

- to access a struct's members, use ***dot notation***:

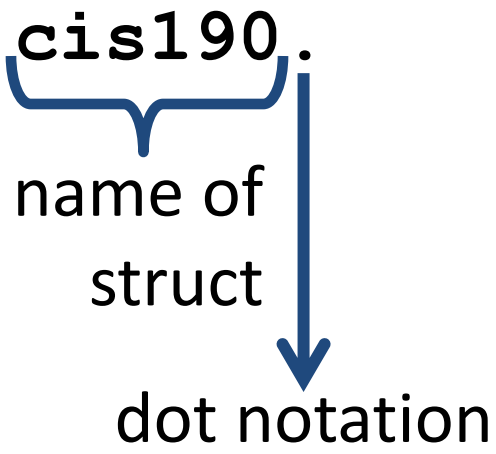


Using Structs

- to declare a variable of type struct cisClass:

```
struct cisClass cis190;
```

- to access a struct's members, use ***dot notation***:

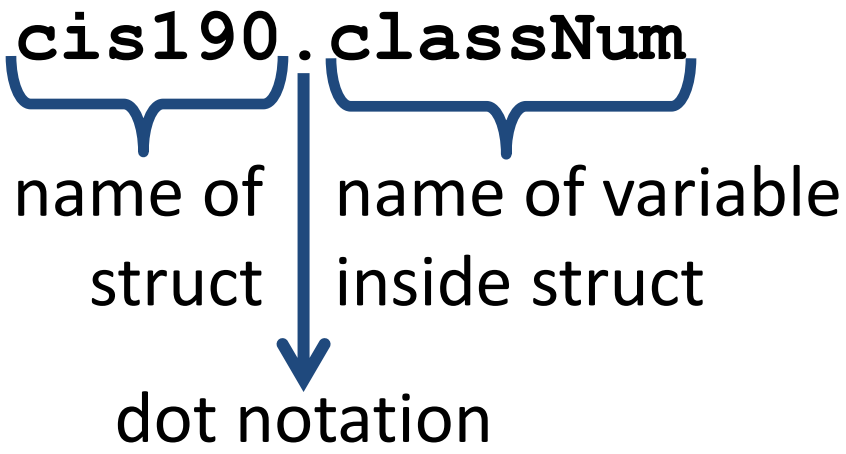


Using Structs

- to declare a variable of type struct cisClass:

```
struct cisClass cis190;
```

- to access a struct's members, use ***dot notation***:

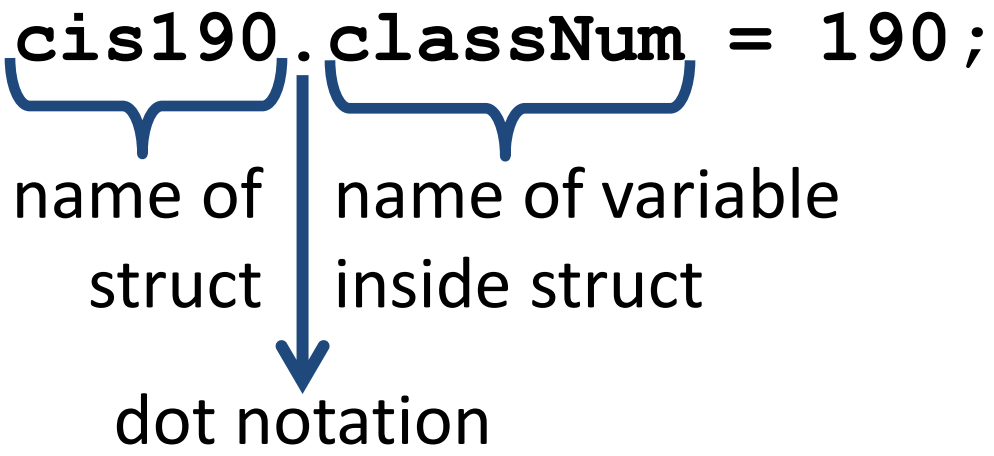


Using Structs

- to declare a variable of type struct cisClass:

```
struct cisClass cis190;
```

- to access a struct's members, use ***dot notation***:



Using Structs

- when using printf:

```
printf("class #: %d\n",
        cis190.classNum);
```

- when using scanf:

```
scanf("%d", &(cis190.classNum) );
```

  - the parentheses are not necessary, but make it clear exactly what we want to happen in the code

typedefs

- ***typedef*** declares an alias for a type

```
typedef unsigned char BYTE;
```

- allows you to refer to a variable by its shorter typedef, instead of the full name

```
unsigned char b1;
```

vs

```
BYTE b2;
```

Using typedefs with Structs

- can use it to simplify struct types:

```
struct cisClass {
   int  classNum;
   char room [20];
   char title [30];
};
```

Using typedefs with Structs

- can use it to simplify struct types:

```
typedef struct cisClass {
   int  classNum;
   char room [20];
   char title [30];
} CIS_CLASS;
```

- so to declare a struct, the code is now just

```
CIS_CLASS cis190;
```

Structs as Variables

- we can treat structs as variables (*mostly*)
  - pass to functions
  - return from functions
  - create arrays of structs
  - and more!
- but we cannot:
  - assign one struct to another using the = operator
  - compare structs using the == operator

Arrays of Structures

```
CIS_CLASS classes [4];
```

classNum	classNum	classNum	classNum
room	room	room	room
title	title	title	title
0	1	2	3

Arrays of Structures

```
CIS_CLASS classes [4];
```

classNum	classNum	classNum	classNum
room	room	room	room
title	title	title	title
0	1	2	3

- access like you would any array:

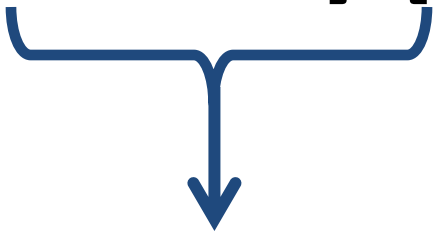
Arrays of Structures

```
CIS_CLASS classes [4];
```

classNum	classNum	classNum	classNum
room	room	room	room
title	title	title	title
0	1	2	3

- access like you would any array:

```
classes[0]
```



element of array to access

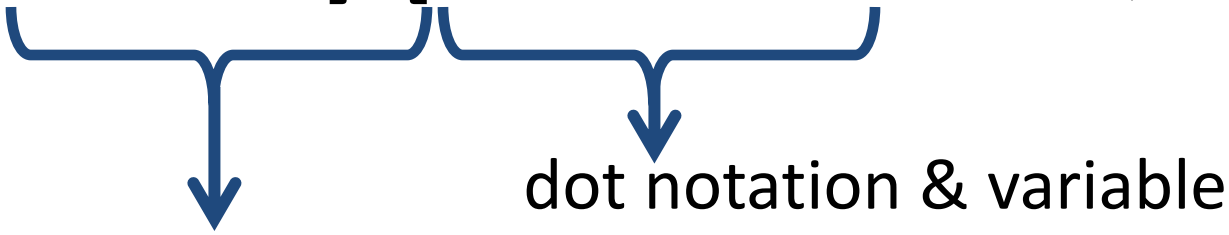
Arrays of Structures

```
CIS_CLASS classes [4];
```

classNum	classNum	classNum	classNum
room	room	room	room
title	title	title	title
0	1	2	3

- access like you would any array:

```
classes[0].classNum = 190;
```



Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

#define

- C's way of creating ***symbolic constants***

```
#define NUM_CLASSES 4
```

- use **`#define`** to avoid "magic numbers"
  - numbers used directly in code

- the compiler replaces all constants at compile time, so anywhere that the code contains **`NUM_CLASSES`** it becomes **`4`** at compile time

#define

- use them the same way you would a variable

```
#define NUM_CLASSES   4
#define MAX_STUDENTS 30
#define DEPARTMENT    "CIS"

CIS_CLASS classes [NUM_CLASSES];

printf("There are %d students allowed in
        %s department mini-courses.\n",
        MAX_STUDENTS, DEPARTMENT);
```

Using #define

- **`#define`** does not take a type
  - or a semicolon

- type is determined based on value given

```
#define FOO   42
```
– integer

```
#define BAR   42.0
```
– double

```
#define H_W "hello"
```
– string

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Pointers

- used to “point” to locations in memory

```
int  x;
int *xPtr;
x = 5;
xPtr = &x;   /* xPtr points to x */
*xPtr = 6;   /* x's value is 6 now */
```

- pointer type must match the type of the variable whose location in memory it points to

Using Pointers with scanf

- remember from last class that scanf uses a pointer for most variable types
  - because it needs to know *where* to store the values it reads in

```
scanf("%d", &int_var);
scanf("%f", &float_var);
```

- remember also that this isn't true for strings:

```
scanf("%s", string_var);
```

Ampersands & Asterisks

- pointers make use of two different symbols
  - ampersand **&**
  - asterisk *****

- ampersand
  - returns the **address** of a **variable**
- asterisk
  - dereferences a **pointer** to get to its **value**

Pointers – Ampersand

- ***ampersand*** returns the address of a variable

```
int  x = 5;
int *varPtr = &x;
int  y = 7;

scanf("%d %d", &x, &y);
```

Pointers – Asterisk

- ***asterisk*** dereferences a pointer to get to its value

```
int  x = 5;
int *varPtr = &x;
int  y = *varPtr;
```

Pointers – Asterisk

- ***asterisk*** dereferences a pointer to get to its value

```
int  x = 5;
int *varPtr = &x;
int  y = *varPtr;
```

- asterisk is also used when initially declaring a pointer (and in function prototypes)

Pointers – Asterisk

- ***asterisk*** dereferences a pointer to get to its value

```
int  x = 5;
int *varPtr = &x;
int  y = *varPtr;
```

- asterisk is also used when initially declaring a pointer (and in function prototypes), but after declaration the asterisk is not used:

```
varPtr = &y;
```

Examples – Ampersand & Asterisk

```
int  x = 5;
int *xPtr;          [* used to declare ptr]

xPtr = &x;          [& used to get address]
                    [but note * is not used]

*xPtr = 10;         [* used to get value]

scanf("%d",&x);  [use & for address]
```

Visualization of pointers

variable name			
memory address			
value			

Visualization of pointers

```
int  x = 5;
```

variable name	x		
memory address	0x7f96c		
value	5		

Visualization of pointers

```
int  x = 5;
int *xPtr = &x;
```

variable name	x	xPtr	
memory address	0x7f96c	0x7f960	
value	5	0x7f96c	

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
```

variable name	x	xPtr	
memory address	0x7f96c	0x7f960	
value	5	0x7f96c	

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

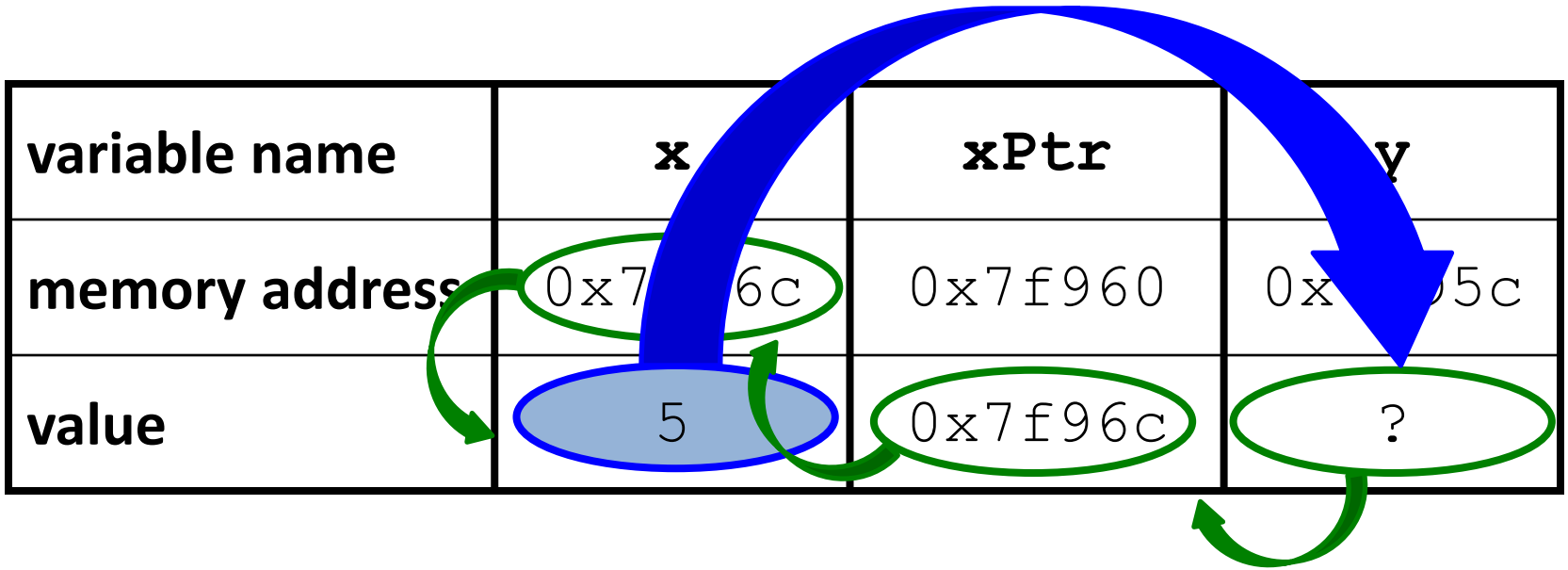
Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	?

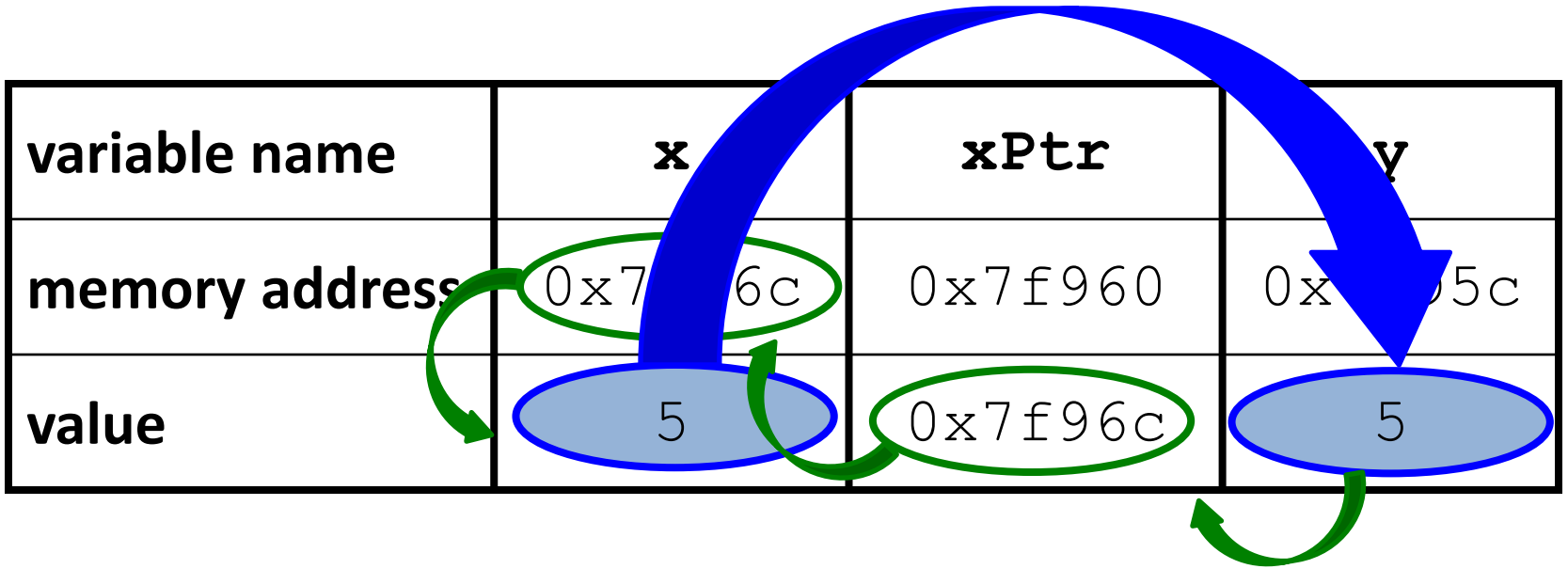
Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is ? */
```



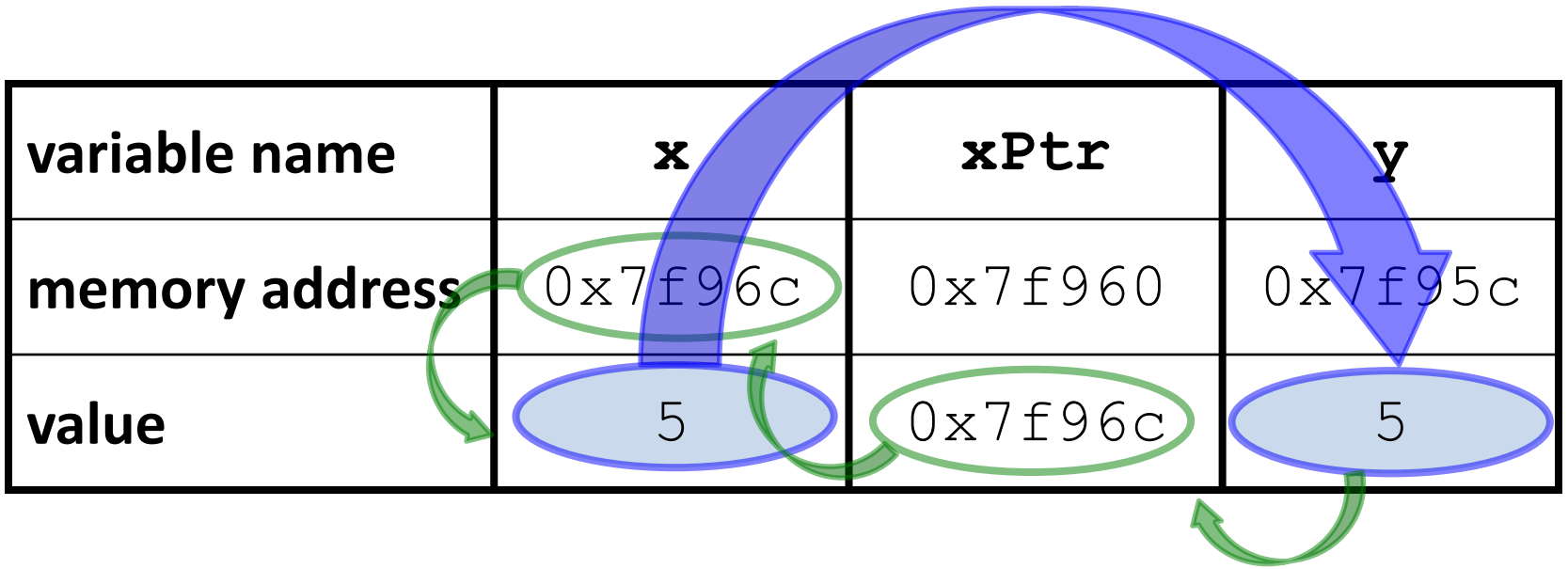
Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is 5 */
```



Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is 5 */
```



variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	5

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is 5 */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	5	0x7f96c	5

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is 5 */
x = 3;          /* y is still 5 */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	3	0x7f96c	5

Visualization of pointers

```
int  x = 5;
int *xPtr = &x; /* xPtr points to x */
int  y = *xPtr; /* y's value is 5 */
x = 3;          /* y is still 5 */
y = 2;          /* x is still 3 */
```

variable name	x	xPtr	y
memory address	0x7f96c	0x7f960	0x7f95c
value	3	0x7f96c	2

Pointer Assignments

- pointers can be assigned to one another using =

```
int  x = 5;
int *xPtr1 = &x; /* xPtr1 points
                    to address of x */
int *xPtr2;      /* uninitialized */

xPtr2 = xPtr1;   /* xPtr2 also points
                    to address of x */
(*xPtr2)++;      /* x is 6 now */
(*xPtr1)--;      /* x is 5 again */
```

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Passing Variables

- when we pass variables like this:

```
int x = 5;
AddOne(x);
```

what happens to **x**?

Passing Variables

- when we pass variables like this:

```
int x = 5;
AddOne(x);
```

  a copy of **x** is made, and the changes made in the function are made to the copy of **x**

- the changes we make to **x** while inside the **AddOne()** function won't be reflected in the "original" **x** variable

Passing Variables

- using pointers allows us to ***pass-by-reference***
  – so we're passing a pointer, not making a copy

- if we pass a variable like this:

  ```
  AddOne(&x);
  ```

  what we are passing is the address where **x** is stored in memory, so the changes made in the function are made to the "original" **x**

Two Example Functions

pass-by-value:

```
void AddOneByVal (int  x) {
  /* changes made to a copy */
  x++;
}
```

pass-by-reference:

```
void AddOneByRef (int *x) {
  /* changes made to "original" */
  (*x)++;
}
```

Two Example Functions

```
int  x = 5;
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);
```

variable name	x	x (copy)
memory address	0x7fa80	0x7fa8c
value	5	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);
```

```
void AddOneByVal (int  x) {
   x++;
}
```

variable name	x	x (copy)
memory address	0x7fa80	0x7fa8c
value	5	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);
```

```
void AddOneByVal (int  x) {
  x++;
}
```

variable name	x	x (copy)
memory address	0x7fa80	0x7fa8c
value	5	6

Two Example Functions

```
int  x = 5;
AddOneByVal(x);
```

```
void AddOneByVal (int  x) {
   x++;
}
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);  /* x = 5 still */
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

```
void AddOneByRef (int *x) {
   (*x)++;
}
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

```
void AddOneByRef (int *x) {
   (*x)++;
}
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

```
void AddOneByRef (int *x) {
   (*x)++;
}
```

variable name	x
memory address	0x7fa80
value	5

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

```
void AddOneByRef (int *x) {
   (*x)++;
}
```

variable name	x
memory address	0x7fa80
value	6

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);
```

```
void AddOneByRef (int *x) {
   (*x)++;
}
```

variable name	x
memory address	0x7fa80
value	6

Two Example Functions

```
int  x = 5;
AddOneByVal(x);   /* x = 5 still */
AddOneByRef(&x);  /* x = 6 now */
```

variable name	x
memory address	0x7fa80
value	6

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Pointers and Arrays

- arrays are pointers!
  - they're pointers to the beginning of the array

- but they are also *only* pointers
- which is why there's
  - no bounds checking
  - no way provided to determine length

Pointers and Arrays and Functions

- because arrays are pointers, they are **always** passed by reference to a function

- this means:
  - the program does not make a copy of an array
  - any changes made to an array inside a function will remain after the function exits

Pointers and Arrays

- passing **one element** of an array is still treated as pass-by-value

  `classes[0]` is a single variable of type **`CIS_CLASS`**, not a pointer to the array

  `intArray[i]` is a single variable of type **`int`**, not a pointer to the array

C-style Strings

- reminder: C strings are **arrays** of characters
  - so functions always pass strings by reference

- remember scanf?

```
scanf("%d", &x);   /* for int */
scanf("%s", str); /* for string */
```

  - there is no "**&**" because C strings are arrays, so scanf is already seeing an address

C-style Strings in Functions

- using in functions:

```
/* function takes a char pointer */
void ToUpper (char *word);
char str[] = "hello";
ToUpper (str);
```

- this is also a valid function prototype:

```
void ToUpper (char word[]);
```

Pointers and Struct Members

- remember, to access a struct's member:

```
cisClass.classNum = 190;
```

- when we are using a pointer to that struct, both of the following are valid expressions to access the member variables:

```
(*cisClassPtr).classNum = 191;
cisClassPtr->classNum = 192;
```

Pointers and Struct Members

- the `->` operator is simply shorthand for using `*` and `.` together
  - the asterisk dereferences the struct so we can access it values, i.e., its member variables
  - the member variables are stored directly in the struct (not as pointers), so we can access them via dot notation, without needing to dereference

```
(*cisClassPtr).classNum = 191;
  cisClassPtr->classNum = 192;
```

Coding Practice

- download starter files from the class website
  - http://www.seas.upenn.edu/~cis190/fall2014

- will use structs to get some practice with
  - pointers
  - arrays
  - passing by reference

LIVECODING

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Makefiles

- use to automate tasks related to programming
  - compiling program
  - linking .o files
  - deleting files
  - running tests
- using a Makefile helps
  - prevent human error
  - facilitate programmer laziness

Makefile Basics

- must be called **`Makefile`** or **`makefile`**
- contains a bunch of rules expressed as:

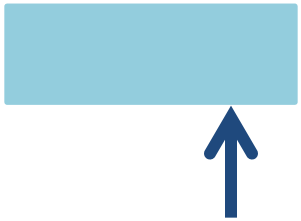
```
target: dependency list
        action
```

- invoke a rule by typing "**`make target`**" in the command line

Makefile Basics

- must be called **`Makefile`** or **`makefile`**
- contains a bunch of rules expressed as:

```
target: dependency list
        action
```



this must be a tab, or it won't work

- invoke a rule by typing "**`make target`**"
  - while in the folder containing the Makefile

Makefile Basics

- comments are denoted by a pound **#** at the beginning of the line

- the very first rule in the file will be invoked if you type "**`make`**" in the command line

- there's a lot of automation you can add to Makefiles – look for more info online

Makefile Basics

- example Makefile on page for Homework 2
  - more info in the Makefile's comments

- Makefiles will be required for all future programming homeworks
  - the first rule in the Makefiles you submit **must** fully compile and link your program
  - graders will use this to compile your program

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Testing

- unit testing
  - literal tests to make sure code works as intended
  - e.g., **`TwoPlusTwoEqualFour(...)`** for an **`Addition()`** function

- ***edge case*** testing (or corner case, etc.)
  - ensure that code performs correctly with all (or at least many) possible input values
  - e.g., prevent program from accepting invalid input

Simple Testing Example

```
/* get month from user in integer form */
printf("Please enter month: ");
scanf("%d", &month);
```

Simple Testing Example

```
/* get month from user in integer form */
printf("Please enter month: ");
scanf("%d", &month);
while (month < JAN_INT || month > DEC_INT)
{



  scanf("%d", &month);
}
```

Simple Testing Example

```
/* get month from user in integer form */
printf("Please enter month: ");
scanf("%d", &month);
while (month < JAN_INT || month > DEC_INT)
{
  printf("\n%d is an invalid month", month);
  printf("please enter between %d and %d:",
         JAN_INT, DEC_INT);
  scanf("%d", &month);
}
```

```
/* print string up to number given
   by length (or full string,
   whichever is reached first) */
void PrintToLength(char str[],
                   int  length)
{
  int i;
  for (i = 0; i < length; i++)
  {
    printf("%c", str[i]);
  }
}
```

Common Edge Cases

- C-style string
  - empty string
  - pointer to NULL
  - without the \0 terminator
- Integer
  - zero
  - negative/positive
  - below/above the min/max

Outline

- Separate Compilation
- Structures
- #define
- Pointers
  - Passing by Value vs. Passing by Reference
  - Pointers and Arrays and Functions and Structs
- Makefiles
- Testing
- Homework

Homework 2

- Trains
  - most difficult part of the homework is formatting the printing of the train cars!
  - make sure output is readable (see sample output)

- hw2.c, trains.c, trains.h (and answers.txt)
  - don't submit the Makefile or any other files!
  - take credit for your code!