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

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

int main()
{
    [...]  
}

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

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

hw2.c
```
Example: Homework 2 Files

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

int main()
{
    [...] 
}

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

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

hw2.c

trains.h

trains.c
Example: Homework 2 Files

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

int main()
{
    [...] 
}

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

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

hw2.c

trains.h

trains.c
```
Example: Homework 2 Files

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

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

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

hw2.c

```c
trains.h
```

```c
trains.c
```
Example: Homework 2 Files

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

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

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

hw2.c

trains.h

trains.c
```
Example: Homework 2 Files

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

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

trains.h

trains.c
```

```c
hw2.c
```
Example: Homework 2 Files

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

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

`hw2.c`

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

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

`trains.h`

`trains.c`
Example: Homework 2 Files

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

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

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

- [hw2.c](#)
- [trains.h](#)
- [trains.c](#)
Example: Homework 2 Files

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

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

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

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

```c
trains.h
```

```c
trains.c
```
Example: Homework 2 Files

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

---

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

---

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

---

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

---

```c
trains.h
```

---

```c
trains.c
```
Example: Homework 2 Files

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

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

trains.c

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

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

Example: Homework 2 Files

```c
#include "trains.h"

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

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

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

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

trains.c
```
#include "trains.h"

int main()
{
    [...] 
}

hw2.c

#include "trains.h"

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

trains.h

#include "trains.h"

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

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

trains.c
Example: Homework 2 Files

```c
#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 \texttt{gcc} and the \texttt{-c} flag
  – linked together
    • using \texttt{gcc} and the created \texttt{.o} (object) files
Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] }

#include "trains.h"

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

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

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

```c
hw2.c

trains.c

trains.h
```
Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] }

    hw2.c
```

```c
#include "trains.h"

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

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

    trains.c
```

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

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

    trains.h
```
Compiling Multiple .c Files

```
#include "trains.h"

int main()
{
    [...]
}

#include "trains.h"

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

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

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

tells the compiler we’re compiling separately

– stops before linking
– won’t throw an error if everything’s not available
#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
```c
#include "trains.h"

int main()
{
    [...] } 

hw2.c

#include "trains.h"

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

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

trains.c

***OBJECT FILE***

hw2.o

> gcc -c -Wall hw2.c

```
Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] }

hw2.c
```

```c
#include "trains.h"

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

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

trains.c
```

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

hw2.o
```

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

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

trains.h
```
Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] 
}

hw2.c

#include "trains.h"

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

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

trains.c

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

```
Compiling Multiple .c Files

```c
#include "trains.h"

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

#include "trains.h"

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

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

trains.c

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

trains.h
```

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

***OBJECT FILE***

hw2.o
```
Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] }

#include "trains.h"

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

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

void PrintTrain(...);

void AddTrainCar(...);

trains.h

trains.c

***OBJECT FILE***

hw2.o

***OBJECT FILE***

trains.o

> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
```
# Compiling Multiple .c Files

```c
#include "trains.h"

int main()
{
    [...] }

hw2.c

#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
```
Linking Multiple .o Files

```c
#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
#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

```c
#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

***EXECUTABLE***
    a.out

> gcc -c -Wall hw2.c
> gcc -c -Wall trains.c
> gcc -Wall hw2.o trains.o
Linking Multiple .o Files

```c
#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

***EXECUTABLE***

a.out

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

trains.o

34
Naming Executables

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

```bash
> gcc -Wall hw2.o trains.o
```
becomes

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

• and to run it, you just type

```bash
> ./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:

```c
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:

```c
struct cisClass cis190;
```

• to access a struct’s members, use *dot notation*:

```
cis190
```
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:

```c
struct cisClass cis190;
```

• to access a struct’s members, use *dot notation*:

```
cis190.classNum
```

(name of struct) (name of variable inside struct) (dot notation)
Using Structs

• to declare a variable of type struct cisClass:
  ```
  struct cisClass cis190;
  ```

• to access a struct’s members, use `dot notation`:
  ```
  cis190.classNum = 190;
  ```
Using Structs

• when using printf:

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

• when using scanf:

```c
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:

```c
struct cisClass {
    int classNum;
    char room [20];
    char title [30];
};
```
Using typedefs with Structs

• can use it to simplify struct types:

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

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

```c
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

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

<table>
<thead>
<tr>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>room</td>
<td>room</td>
<td>room</td>
<td>room</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Arrays of Structures

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

<table>
<thead>
<tr>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>room</td>
<td>room</td>
<td>room</td>
<td>room</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
<td>title</td>
<td>title</td>
</tr>
</tbody>
</table>

• access like you would any array:
Arrays of Structures

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

<table>
<thead>
<tr>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
<th>classNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>room</td>
<td>room</td>
<td>room</td>
<td>room</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
<td>title</td>
<td>title</td>
</tr>
</tbody>
</table>

- access like you would any array:

```
classes[0]
```

element of array to access
Arrays of Structures

```c
CIS_CLASS classes [4];

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>room</td>
<td>room</td>
<td>room</td>
<td>room</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
<td>title</td>
<td>title</td>
</tr>
</tbody>
</table>
```

- access like you would any array:
  ```c
classes[0].classNum = 190;
  ```
  dot notation & variable
  element of array to access
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
• use them the same way you would a variable

```c
#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

  ```c
  #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

```c
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

```c
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

```c
int x = 5;
int *varPtr = &x;
int y = *varPtr;
```
Pointers – Asterisk

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

```c
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

```c
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:

```c
varPtr = &y;
```
Examples – Ampersand & Asterisk

```c
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

<table>
<thead>
<tr>
<th>variable name</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int x = 5;

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>?</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>5</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>5</td>
</tr>
</tbody>
</table>
Visualization of pointers

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>0x7f96c</td>
<td>5</td>
</tr>
</tbody>
</table>
int x = 5;
int *xPtr = &x; /* xPtr points to x */
int y = *xPtr; /* y’s value is 5 */
x = 3; /* y is still 5 */
Visualization of pointers

```c
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 */
```

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>xPtr</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7f96c</td>
<td>0x7f960</td>
<td>0x7f95c</td>
</tr>
<tr>
<td>value</td>
<td>3</td>
<td>0x7f96c</td>
<td>2</td>
</tr>
</tbody>
</table>
Pointer Assignments

• pointers can be assigned to one another using =

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

  what happens to \textbf{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:

```c
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:**

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

**pass-by-reference:**

```c
void AddOneByRef (int *x) {
    /* changes made to “original” */
    (*x)++;
}
```
Two Example Functions

```c
int x = 5;
```

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>x (copy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
<td>0x7fa8c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>x (copy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
<td>0x7fa8c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
<th>x (copy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
<td>0x7fa8c</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>6</td>
</tr>
</tbody>
</table>
Two Example Functions

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

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>6</td>
</tr>
</tbody>
</table>
Two Example Functions

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

<table>
<thead>
<tr>
<th>variable name</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory address</td>
<td>0x7fa80</td>
</tr>
<tr>
<td>value</td>
<td>6</td>
</tr>
</tbody>
</table>
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

  \[
  \text{classes[0]} \quad \text{is a single variable of type} \\
  \text{CIS\_CLASS, not a pointer to the array}
  \]

  \[
  \text{intArray[i]} \quad \text{is a single variable of type} \\
  \text{int, not a pointer to the array}
  \]
C-style Strings

• reminder: C strings are arrays of characters
  – so functions always pass strings by reference

• remember scanf?
  
  ```c
  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:

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

• this is also a valid function prototype:

```c
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 \texttt{\rightarrow} operator is simply shorthand for using \texttt{*} and \texttt{.} together
  – the asterisk dereferences the struct so we can access its 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

\begin{verbatim}
(*cisClassPtr).classNum = 191;
cisClassPtr->classNum = 192;
\end{verbatim}
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
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 \texttt{Makefile} or \texttt{makefile}

• contains a bunch of rules expressed as:
  \begin{verbatim}
  target: dependency list
  \end{verbatim}

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

• invoke a rule by typing "\texttt{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., \texttt{TwoPlusTwoEqualFour(...)} for an \texttt{Addition()} function

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

/* 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!