CIS 190: C/C++ Programming

Lecture 2

Pointers and More

Separate Compilation

- to prevent the file containing main() from getting too crowded and long
  - function prototypes in their own file (functions.h)
  - function definitions in their own file (functions.c)

- put **`#include "functions.h"`** at top of any .c file using those functions
  - note that we use quotes (“) instead of carats (<>)
- need to compile files separately

Compiling with multiple .c files

- for three files: main.c, functions.c, functions.h
  - main.c and functions.c both have
    `#include "functions.h"`

```
> gcc -c -Wall main.c
> gcc -c -Wall functions.c
> gcc -Wall main.o functions.o
      -o main
```

Separate Compilation Mistakes

- Don't #include **.c files**
- Don't put #include **in** a .h file
- Only #include those files whose function prototypes are needed

- getting the error: "**`undefined reference to 'functionName'`**"
  - linker couldn't find the function 'functionName'
  - 99% of the time, this is because 'functionName' was spelled wrong somewhere

Structures

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

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

Using Structures

- to declare a structure of type cisClass:

```
struct cisClass cis190;
```

- to access a variable inside, use dot notation:

```
cis190.classNum = 190;
strcpy(cis190.room, "Towne 309");
printf("class #: %d\n",
cis190.classNum);
```

- when using scanf:

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

typedefs

- typedef declares an alias for a type

```
typedef unsigned char uchar;
```

- can use it to simplify struct types:

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

Arrays of Structures

- structures are variables, which means we can make arrays of them:

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

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

- access like an array:

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

#define

- symbolic constants – replaced at compile time

```
#define NUM_CLASSES 4
```

- use #define to avoid “magic numbers”
  – numbers used directly in code

- used the same way you would a variable

```
CIS_CLASS classes [NUM_CLASSES];
```

Pointers

- “point” to locations in memory

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

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

- scanf uses pointers for ints, etc. because it needs to know where to store the values it reads in

```
scanf(“%d”, &x);
```

Accessing data in pointers

- & - ampersand; returns the address of a **value**

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

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

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

Visualization of pointers

variable			
memory address			
value			

Visualization of pointers

```
int  x = 5;      /* x = 5 */
```

variable	x		
memory address	0x7f96c		
value	5		

Visualization of pointers

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

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

Visualization of pointers

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

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

Visualization of pointers

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

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

Visualization of pointers

```
int  x = 5;       /* 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 = 3 and y = 2 */
```

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

Pointer Assignments

- pointers can be assigned to one another:

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

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

Pointers and functions

- pointers allow us to **call-by-reference**
  - previously we could only call-by-value

- passing by reference allows the variable to be changed inside the function:

```
void AddOneByVal (int  var) {   var++; }
void AddOneByRef (int *var) { (*var)++; }
```

- calling functions with pointers

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

Pointers and functions

```
int x = 5;
printf("x at    start:        %d\n", x);
AddOneByVal(x);
printf("x after AddOneByVal: %d\n", x);
AddOneByRef(&x);
printf("x after AddOneByRef: %d\n", x);
```

```
> x at    start:        5
> x after AddOneByVal: 5
> x after AddOneByRef: 6
```

Pointers and arrays

- arrays are pointers!
  - they’re pointers to the first element in the array
- arrays are not exactly pointers!
  - **cannot** assign one array to another

- this results in a syntax error:

```
array1 = array2;
```

Pointers and arrays and functions

since arrays are pointers, that means:

- arrays passed to a function **always** result in call-by-reference
  - does not make a copy of the array
  - any changes made to an array in a function will remain
- passing ONE ELEMENT is still call-by-value
  - **`classes[0]`** is a value, not a pointer

Pointers and structs

- remember, to access a structure member:

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

- when we are using a pointer to that struct:

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

- the **->** operator is simply shorthand for using ***** and **.** together
  - to access the value of a member of a structure

C-style strings are arrays too

- reminder: C strings are **arrays** of characters
  - so use in functions is always call-by-reference

- remember scanf?

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

- no "**&**" because C-strings are arrays

C-style strings in functions

- using in functions:

```
/* function takes char pointer */
void ToUpper (char *word);

char* str = "hello"; /* c string*/

/* str is a ptr to an array of chars*/
ToUpper (str);
```

Makefiles

- contain a list of rules called by typing

  `make ruleName`

  in the command line
- example Makefile on the page for HW2
  - more info in the comments inside the Makefile
  - can create your own rules
  - makes compiling, etc. a lot quicker and easier

Homework 2

- Trains
  - structs, arrays of structs, C strings, separate compilation, printf formatting, pointers
  - hardest part is printing the train!
  - readability of output (see sample output)

- hw2.c, trains.c, trains.h
  - don’t submit Makefile or any other files!
  - take credit for your code!