CSE 333 – SECTION 4

Makefiles, FSMs, vowely-words

Some slides referenced from CSE 333 Autumn 2020

Logistics

Thursday, Jan 28 (tonight!)

- HW1 due 11pm

Section Plan

- Enums
- Function pointers
- FSM definition
- Makefiles

Enums

- User-defined types that can have one of a specified list of values “one-of types”
- Syntax:

```
enum <type_name> { <option_1>, …, <option_n>};

enum <type_name> <var_name> = <option>;
```

- Ex:

```
enum month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
                SEP, OCT, NOV, DEC };

enum month cur_month;

cur_month = JAN;
```

Enums (cont)

- Similar to structs, the type of an enum is enum <type_name> (eg: cur_month is a variable of type ‘enum month’)
- Can use similar typedef “trick” to structs:

```
typedef enum month month_t;

month_t next_month = FEB;
```

What actually *is* an enum?

- Each option in list assigned an int
  - By default, counts in order from 0, but you can specify different values

```
enum food { SOUP = 3, SAND = 4, SODA = 1 };
```

- Can be used to declare constants similar to #define
  - Has some benefits because it’s not a preprocessor command (scope, etc)
  - Also allows you to declare multiple at a time

Function Pointers

- Recall from 351: function code is stored in memory, so we can create a pointer to the location where that code is stored
- Syntax:

```
declaration: <return_type> (* <name>) (<type_1>, ..., <type_n>)

call: (* <name>) (<arg_1>, ..., <arg_n>)
```

- Uses:
  - Can pass functions pointers around as variables/parameters,
    - sending custom code to callee, or
    - leaving implementation to caller
  - Code reuse
  - “Generics” in C (HW1)
  - “Callbacks” - allow some other function to be triggered by a particular event

Function Pointer Example

```
int sum(int x, int y) {
      return x + y;
}
int mult(int x, int y) {
      return x * y;
}
int apply_to_array(int* arr, int length,
                        int (*fun)(int, int)) {
      int total = *arr;
      for (int i = 1; i < length; i++) {
            total = fun(total, *(arr + i));
      }
      return total;
}
```

```
int main(int argc, char** argv) {
      int arr[] = {2, 3, 4};
      int sum_total = apply_to_array(arr, 3, sum);
      int mult_total = apply_to_array(arr, 3, mult);
      printf("sum = %d, mult = %d", sum_total,
                  mult_total);
}
```

Output:

Function Pointer Example

```
int sum(int x, int y) {
      return x + y;
}
int mult(int x, int y) {
      return x * y;
}
int apply_to_array(int* arr, int length,
                        int (*fun)(int, int)) {
      int total = *arr;
      for (int i = 1; i < length; i++) {
            total = fun(total, *(arr + i));
      }
      return total;
}
```

```
int main(int argc, char** argv) {
      int arr[] = {2, 3, 4};
      int sum_total = apply_to_array(arr, 3, sum);
      int mult_total = apply_to_array(arr, 3, mult);
      printf("sum = %d, mult = %d", sum_total,
                  mult_total);
}
```

Output:
sum = 9, mult = 24

Function Pointer Example (qsort from cplusplus.com)

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));
```

```
/* qsort example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort */

int values[] = { 40, 10, 100, 90, 20, 25 };

int main ()
{
  int n;
  qsort (values, 6, sizeof(int), less_than);
  for (n = 0; n < 6; n++)
     printf ("%d ",values[n]);
  return 0;
}
```

```
int less_than (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}
```

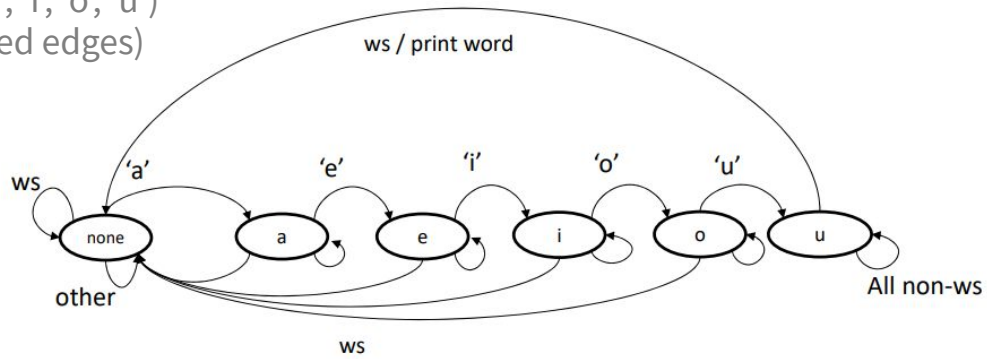
Suppose now we want to sort in descending order:

```
int greater_than (const void * a, const void * b)
{
  return ???;
}
```

Notice the use of void for generics! Similar to HW1.

Finite State Machines (FSMs)

- Can represent a program as a series of states with ways to move between them
- A model of computation (with limited amount of memory) consisting of
  - finite set of states Q (none, a, e, i, o, u)
  - finite set of characters Σ (‘ws’, ‘cons’, ‘a’, ‘e’, ‘i’, ‘o’, ‘u’)
  - transition function $\delta : Q \times \Sigma \rightarrow Q$ (the labeled edges)
  - a start state $q0 \in Q$ (none)
  - set of accepted states $F \subseteq Q$
- Execution by changing states q:
  - start at start state $q = q0$
  - Each step
    - reads a character c as input
    - transitions to state $q = \delta(q, c)$
  - Upon end of input, output accept if q is a accepted state ($q \in F$); reject otherwise



- Less powerful than most programming language
- Useful for reducing logical complexity in certain applications
  - e.g. stream processing, formal languages, managing states in distributed systems
  - sorting out the state machine helps even just conceptually

Representing FSMs in Data

Input Character

Current State

	a	**e**	**i**	**o**	**u**	**cons**	**ws**
none	a	none	none	none	none	none	none
a	none	e	none	none	none	a	none
e	none	none	i	none	none	e	none
i	none	none	none	o	none	i	none
o	none	none	none	none	u	o	none
u	none	none	none	none	none	u	none

Makefiles Exercise (page 5 on worksheet)

Point.h

```
class Point { … };
```

UsePoint.cc

```
#include "Point.h"
#include "Thing.h"
int main( … ) { … }
```

UseThing.cc

```
#include "Thing.h"
int main( … ) { … }
```

Point.cc

```
#include "Point.h"
// defs of methods
```

Thing.h

```
struct Thing { … };
// full struct def here
```

Alone.cc

```
int main( … ) { … }
```

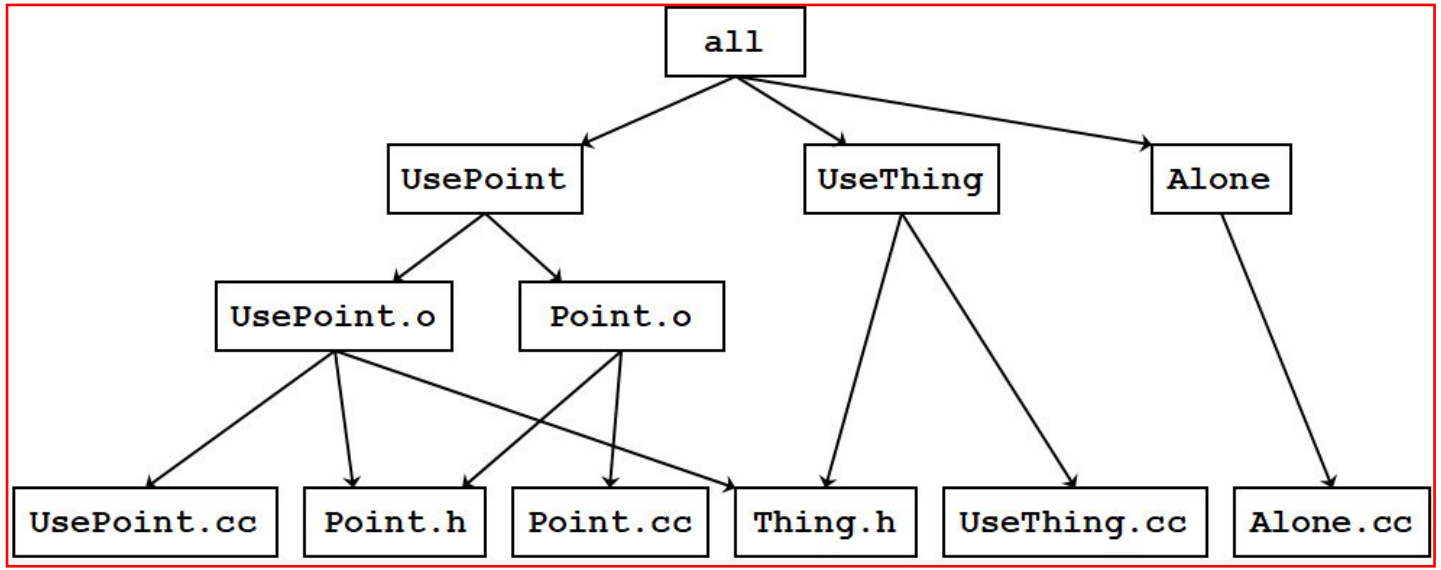
DAG

Point.h	`class Point { … };`

UsePoint.cc	`#include "Point.h"`
	`#include "Thing.h"`
	`int main( … ) { … }`

UseThing.cc	`#include "Thing.h"` `int main( … ) { … }`

Point.cc	`#include "Point.h"` `// defs of methods`
Thing.h	`struct Thing { … };` `// full struct def here`
Alone.cc	`int main( … ) { … }`



Makefile

```
CFLAGS = -Wall -g -std=c++11

all: UsePoint UseThing Alone

UsePoint: UsePoint.o Point.o
    g++ $(CFLAGS) -o UsePoint UsePoint.o Point.o

UsePoint.o: UsePoint.cc Point.h Thing.h
    g++ $(CFLAGS) -c UsePoint.cc

Point.o: Point.cc Point.h
    g++ $(CFLAGS) -c Point.cc

UseThing: UseThing.cc Thing.h
    g++ $(CFLAGS) -o UseThing UseThing.cc

Alone: Alone.cc
    g++ $(CFLAGS) -o Alone Alone.cc

clean:
```