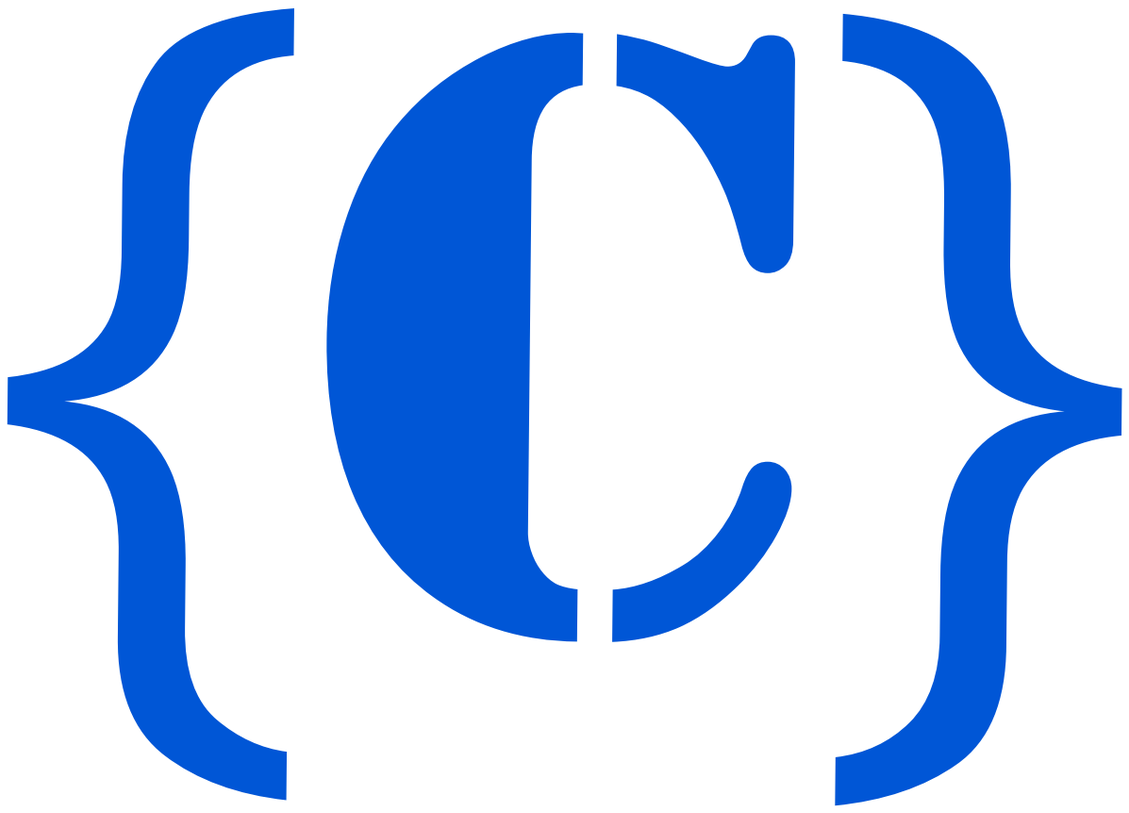
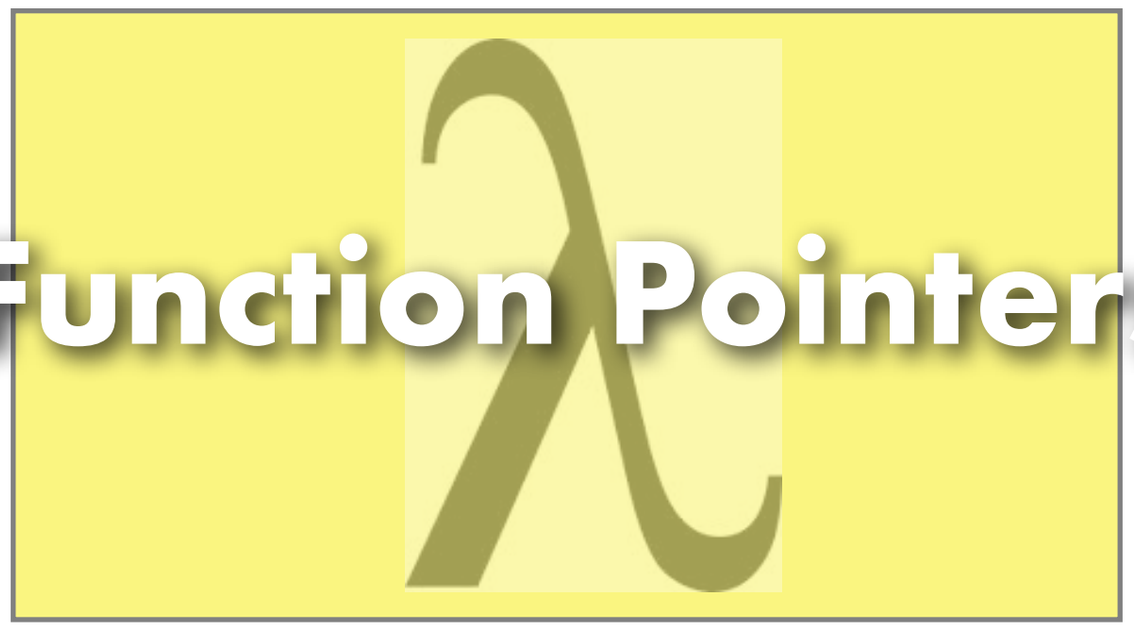


Lecture 15



Abstraction

How are abstractions manifested in languages?

- As structures that encapsulate code and data providing information hiding

  E.g. Classes in Java

- As program structures that refactor common usage patterns

  E.g., a sorting routine that can sort lists of different types

C doesn't provide data abstractions like Java classes

- No obvious way to package related data & code within a single structure

But, it does a provide a useful refactoring mechanism

- Functions are the most obvious example
- They abstract a computation over input arguments
- What kinds of arguments can these be?

Types and Computation

Functions can be abstracted over

- basic types (e.g., int, float, double,...)
- structured types (e.g., structs, unions, ...)

These types are primitive data abstractions

- They represent a set of values along with operations on them

What about functions themselves?

- They're obviously a form of abstraction
- Rather than representing a set of values, they represent a set of computations abstracted over arguments of a fixed type
- There is exactly one operation allowed on function types: application

Types

Following this line of thought:

- A type is a set of values equipped with a set of operations on those values
- A function is a computation abstracted over the types defined by its inputs
- Hence, a function is an abstraction: it represents the set of values produced by its computation when instantiated with specific arguments.

Hence, functions should be allowed to be abstracted over functions, just as they are abstracted over primitives & structures

Concretely…

C permits functions to be treated like any other data object

- A function pointer can be supplied as an argument
- Returned as a result
- Stored in any array
- Compared

Main caveat:

- Cannot deference the object pointed to by a function pointer on the left-hand side of an assignment

Example

Operating on a list of integers…

```
struct list {
        int val;
        struct list *next;
};

typedef struct list List;
```

Creating lists...


```
#include <stdio.h>
List *makeList(int n) {
    List *l, *l1 = NULL;
    for (int i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->val  = n-i;
        l->next = l1;
        l1 = l;
    }
    return l;
}
```

Given a number n, build a list of length n containing elements 1..n inclusive

Creating lists... (2)


```
#include <stdio.h>
List *makeList(int n) {
    List *l, *l1 = NULL;
    for (int i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->val  = i+1;
        l->next = l1;
        l1 = l;
    }
    return l;
}
```

Given a number n, build a list of length n where the ith element of the list contains n-i+1

Creating lists… (3)

We can imagine many different ways of populating a list

- The overall control structure remains the same
- Only the computation responsible for producing the next element changes

How can we abstract the definition to reuse the same control structure for the different kinds of lists we might want?

Function pointers

Supply a pointer to the function that computes values

```
int add (int m) {
    static int n = 0;
    n++;
    return m+n;
}

int min(int m) {
    static int n = 0;
    n++;
    return (m < n) ? m : n;
}
```

Would like to abstract the construction of lists so they can be *parameterized* over specific operators that determine how the elements of the list should be filled in

Syntax

To use a function as a parameter to another function, provide its type as follows:

```
[return type] (*[name])([parameters])
```

Abstraction revisited


```
List *makeGenList (int n, int (*f)(int)) {
    List * l, *l1 = NULL;
    for (int i = 0; i < n; i++) {
        l = (List*) malloc(sizeof(List));
        l->val  = (*f)(i);
        l->next = l1;
        l1 = l;
    };
    return l;
}
```

Expects a function pointer that points to a function which yields an int, and which expects an int argument

Applies (invokes) the function pointed to by f with argument n

```
makeGenList(10,min);
 // { 9, 8, 7, … }
makeGenList(10,add);
 // {19, 17, 15, …}
```

Can create lists with different elements (but same structure) without changing underlying implementation

Deriving a Recipe

- Suppose we want a generic way to perform some computation over all the elements in an integer list.
  - Examples: sum all elements, find min/max of all elements, etc.
  - *Key requirement*: the computation can be expressed in terms of a given list element, and the intermediate computation produced thus far
- Start the computation with an initial value for the result (call this the `accumulator')
- Supply a function pointer that represents the computation.
  - Its signature is fixed - it takes two integers:
    - one for the list element that is being examined as we iterate over the list
    - one for the accumulator
- Use the result of applying the function as the new accumulated value
- Repeat until all list elements have been examined

Next step…

Now, lets define abstractions that compute over lists

```
int fold(int(*f)(int,int), List *l, int acc) {
    if (l == NULL)
        return acc;
    else {
        int x = l->val;
        return fold (f, l->next, (*f)(x,acc));
    }
}
```

a list of integers

an accumulator

A function pointer that operates over pairs of integers and returns an int

Each recursive call to fold operates on the current value and the current accumulator; the result becomes the new value of the accumulator in the next call

Using fold

Each computation expressed using the same definition

```
int sum(int x,int y){ return x + y; }
int mul(int x,int y){ return x * y; }
int max(int x,int y){ return (x>y)?x:y; }
int main () {
    int s,m,x; List *l;
    l = makeGenList(10, min);
    s = fold(sum,l,0);
    m = fold(mul,l,1);
    x = fold(max,l,0);
}
```

Example


```
l = {9, 8, 7, … }
fold(sum, l, 0) =>
  fold(sum, {9, 8, …}, sum(9, 0)) // l->val = 9, acc = 0 =>
  fold(sum, {8, 7, …}, sum(8, 9) // l->val = 8, acc = 9 =>
  fold(sum, {7, 6, …}, sum(7, 17) // l->val = 7, acc = 17 =>
  fold(sum, {6, 5, …}, sum(6, 24) // l->val = 6, acc = 24 =>
fold(sum, {5, 6, …}, sum(5, 30) // l->val = 5, acc = 30 =>
                    .
                    .
                    .
fold(sum, {1, 0}, sum(1, 44) // l->val = 1, acc = 44 =>
fold(sum, { 0 }, sum(0, 45) // l->val = 0, acc = 45
fold(sum, NULL, 45) return acc // acc = 45
```

Map

Fold allows a function to operate over the elements of a list

- Exercise: replace the recursive implementation with an imperative one

C's type system conspires against richer kinds of operations

- the accumulator must be an int
- one can circumvent these limitations using casts and voids

Instead of accumulating a result, suppose we want to apply a function to each element in the list?

- Such operations are called maps

Map


```
List* map(int(*f)(int), List *l) {
   if (l == NULL) return l;
   List * l1 = malloc(sizeof(List));
   l1->val  = (*f)(l->val);
   l1->next = map( f, l->next);
   return l1;
}
```

A function pointer that points to a function which takes an integer argument and produces an integer result

Apply the function pointed to by f to the current list element

Recursively apply map to the rest of the list

Map


```
int add(int m) { return m+1; }
int min(int m) { return m-1; }
int eve(int x) { return (x%2 == 0) ? 1 : 0; }

int main () {
    int a,m,e;
List *l, *eveL, *addL, *minL;
l = makeGenList(10,...);
eveL = map( eve,l);
addL = map (add,l);
minL = map (min,l);
....
```

Objects…

Typical code that could use an object-oriented solution

```
enum TYPE{SQUARE,RECT,CIRCLE,POLYGON};
struct shape {
    float params[MAX];
    enum TYPE type;
};
typedef struct shape Shape;

void draw(Shape* s) {
    switch(s->type) {
        case SQUARE: draw_square(s); break;
        case RECT:   draw_rect(s); break;
        ...
```

Objects…are arrays of fun *s

Typical code that could use an object-oriented solution

```
void (*fp [4])( Shape* s) = {
    drawSquare, drawRec,
    drawCircle, drawPoly};
```

```
void draw (Shape* s ) {
 (*fp[s->type])(s); // call right fun
}
```

Managing state…

Objects are more than array of fun pointers, they have state

Similarly, some functions need to retain private state

Example: Counter

- implement a function that counts the number of times it was invoked
- the count should be hidden
- there can be multiple counters in the program