CS240: Programming in C

Lecture 11: Function Pointers

Abstractions in Programming

- How are abstractions manifested in languages?
  - As structures that encapsulate code and data providing information hiding
    - E.g., a Java class
  - As program structures that *refactor* common usage patterns
    - E.g., a sorting routine that can sort lists of different types
- The two notions are obviously related
  - public C m1(C’ o) { ... o.m(...) ...}
    - Can be applied to any object of instantiated from class C’ or its *subclasses*
    - The context in which M is applied must be one that expects objects of type C or any of its *superclasses*

Abstractions in C

- C doesn't provide data abstractions like Java classes
    - There is no easy or obvious way to package related data and code within a single structure
        - Hard to enforce information hiding
- 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 can be thought of as 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* (or a data abstraction) 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 the computation it defines when instantiated with specific arguments.
    - Thus, its type is characterized by its argument types and the result of its computation
- Hence, *functions should be allowed to be abstracted over function types*, just as they are allowed to be abstracted over primitive and structure types

Concretely ...

- C permits functions (more accurately, function pointers) 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, etc.
- Main caveat:
  - Cannot deference the object pointed to by a function pointer on the left-hand side of an assignment

Motivation (again)

- Provides a means to abstract more complex forms of computations
  - Computations that are abstract over other computations as well as other data
- Unlike other languages that support function abstraction, C supports this notion in a very restrictive and uninspired way
  - See Scheme, Haskell, ML, ... as examples of languages in which functions are truly first-class
  - How are methods treated in Java? What forms of (if any) of function abstraction does it support?

Example

- We'll consider ways that we can perform operations on a list of integers

```
struct List {
  int node;
  struct List * next;
};
```

Generating a list

- Our first task is to figure out a scheme to populate a list with values

```
struct List *makeList(int n) {
  int i;
  struct List * l;
  struct List * l1 = NULL;
  for (i = 0; i < n; i++) {
    l = malloc(sizeof(struct List));
    l->node = 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

Generating a list (cont)

- Here's another definition

```
struct List *makeList1(int n) {
  int i;
  struct List * l;
  struct List * l1 = NULL;
  for (i = 0; i < n; i++) {
    l = malloc(sizeof(struct List));
    l->node = n-i;
    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 i

Generating a list

- 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 refactor (or abstract) the definition so that we can reuse the same control structure for the different kinds of lists we might want?

Function Pointers

- Supply a function pointer that points to the function responsible for computing the value of list elements

```
int add (int m) {
  static int n = 0;
  n++;
  return m-n+1;
}

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

The expression *add or *minus returns a pointer to the code represented by add and minus, resp.

Abstraction revisited

```
struct List *makeGenList (int n, int (*f)(int)) {
  int i;
  struct List * l;
  struct List * l1 = NULL;
  for (i = 0; i < n; i++) {
    l = malloc(sizeof(struct List));
    l->node = (*f)(n);
    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,(*minus));
makeGenList(10,(*plus))
```

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

Next step...

- Now that we can generate lists that hold different kinds of related values, we define abstractions that compute over lists

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

Each recursive call to fold performs an operation on the current list element and the current accumulator; the result becomes the new value of the accumulator in the next call

Using fold

```
int sum (int x, int y) {
  return x + y;
}

int mult (int x, int y) {
  return x * y;
}

int maximum (int x, int y) {
  if (x > y) { return x; }
  else return y;
}
```

```
int main () {
  int s,m,max;
  struct List *l;
  l = makeGenList(10, (*minus));
  s = fold((*sum),l,0);
  m = fold((*mult),l,1);
  max = fold((*maximum),l,0);
}
```

Each computation (sum, mult, max, ...) expressed using the same definition (fold)

Another Example: map

- Fold allows the expression of a function over the collection of elements defined by the list (e.g., sum, mult, max, ...)
- C’s type system conspires against (obviously) richer kinds of operations
  - The accumulator must be an int
  - Can circumvent the type system using casts (next lecture), but this is quite unsafe
- Instead of accumulating a result based on the collection, suppose we want to apply a function to each element in the list?
  - Such operations are called *maps*

Map

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

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 (cont)

```
 int add (int m) {
   return m+1;
 }

 int minus(int m) {
   return m-1;
 }

int even(int x) {
  if (x%2 == 0)
    { return 1; }
  else { return 0; }
}
```

```
int main () {
  int a,m,e;
  struct List *l, *evList,
              *addList, *minusList;

  l = makeGenList(10,...);

  evList = map( (*even),l);
  addList = map ((*add),l);
  minusList = map ((*minus),l);
  ....
```

Example

```
enum TYPE{SQUARE,RECT,CIRCLE,POLYGON};

struct  shape {
    float params[MAX];
    enum TYPE type;
 };

void draw ( struct shape∗ ps ) {
   switch(ps–>type) {
     case SQUARE: draw_square ( ps ) ; break ;
     case RECT: draw_rect ( ps ) ; break ;
...
```

Arrays of function pointers

```
void (∗fp [4])( struct shape∗ ps) =
    { &draw_square, &draw_rec, &draw_circle ,&draw_poly };
```

which is the same as:

```
void (∗fp [4])( struct shape∗ ps) =
    { (*draw_square), (*draw_rec), (*draw_circle) ,(*draw_poly) };
```

```
void draw ( struct    shape∗ ps ) {
   (∗fp[ps–>type])(ps); /∗ call the correct function∗/
}
```

Counters

Defining a counter:

```
int count1 = 0;
  ....
int countn = 0;

int count (int *x) {
   return ++(*x);
}
```

Not modular: need to define a global variable for each counter

```
int count (int *x) {
   static count = 0;
   return ++(*x);
}
```

Hides the counter variable, but can't generate multiple counters

What’s the problem ...

- A counter generator needs to have its own copy of the counter.
- In Java, a counter generator would be a class whose instances have their own copy of the counter value
- What do we need to do to express similar functionality in C?

Closures

```
typedef void * (*generic_function)(void *, ...);
typedef struct {
   generic_function function;
   void *environment;
} closure;
```

args

environment

To a first approximation, think of a counter object as having two parts - (1) the code that implements the counter, and (2) the “environment” that holds the counter value

Void types

A void type represents a type that has no elements.

A pointer to a void type points to a value that has no type.

This means there are no allowable operations on them.

Need to cast void pointers to a pointer of a concrete type in order to access the target value.

One useful application of void pointers is to pass "generic" parameters to a function

Void types (cont)

```
void f (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}
```

```
int main ()
{
  char a = 'x';
  int b = 1602;
  f (&a,sizeof(a));
  f (&b,sizeof(b));
  return 0;
}
```

*What is the value of *a and *b after the two calls to f?*

Void types

void pointers can be used to point to any data type •

```
int x; void* p=&x; /*points to int */ •
float f;void*p=&f;/*points to float*/
```

• void pointers cannot be dereferenced.
The pointers should always be cast before dereferencing.

```
void*p; printf("%d",*p);/*invalid*/
void* p; int *px=(int*)p; printf ("%d",*px); /*valid */
```

Counters revisited

```
int nextval(void *environment);

closure make_counter(int startval)
{
   closure c;

   int *value = malloc(sizeof(int));
   *value = startval;

   c.function = (generic_function)nextval;
   c.environment = value;

   return c;
}
```

Counter generator

```
int nextval(void *environment)
{
   int *value = environment;

   (*value)++;

   return (*value);
}
```

Using the generator

```
int main()
{
   /* Create the two closures */
   closure my_counter = make_counter(2);
   closure my_other_counter = make_counter(3);

   /* Run the closures */
   printf("The next value is %d\n",
    ((generic_function)my_counter.function)
    (my_counter.environment))
   printf("The next value is %d\n",
     ((generic_function)my_other_counter.function)
     (my_other_counter.environment));
   printf("The next value is %d\n",
     ((generic_function)my_counter.function)
     (my_counter.environment));

   return 0;
}
```