Function Pointers and Abstract Data Types

Prof. David August
COS 217

Goals of Today’s Lecture

• Function pointers
  ◦ Sorting an array of integers
  ◦ Sorting an array of strings
  ◦ Sorting an array of any type
    – Void pointers and casting
    – Pointers to functions
• Abstract Data Types
  ◦ Making “array” an ADT

Sorting an Array of Integers

• Example problem
  ◦ Input: array \( v \) of \( n \) integers
  ◦ Output: array in sorted order, from smallest to largest
• Many ways to sort, but three common aspects
  ◦ Comparison between any two elements
  ◦ Exchange to reverse the order of two elements
  ◦ Algorithm that makes comparisons and exchanges till done
• Simple approach
  ◦ Go one by one through the \( n \) array elements
  ◦ By the end of step \( i \), get \( i^{th} \) smallest value in element \( i \)
    – Compare element \( i \) with all elements after it
    – Swap values if the \( i^{th} \) element is larger
### Integer Sorting Example

<table>
<thead>
<tr>
<th>v[0] &gt; v[1]?</th>
<th>7 2 9 6</th>
<th>v[1] &gt; v[2]?</th>
<th>2 7 9 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, swap</td>
<td>2 7 9 6</td>
<td>v[1] &gt; v[3]?</td>
<td>2 7 9 6</td>
</tr>
<tr>
<td>v[0] &gt; v[2]?</td>
<td>2 7 9 6</td>
<td>v[0] &gt; v[3]?</td>
<td>2 7 9 6</td>
</tr>
</tbody>
</table>

### Integer Sorting Function

```c
void sort(int *v, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i+1; j < n; j++) {
            if (v[i] > v[j]) {
                int swap = v[i];
                v[i] = v[j];
                v[j] = swap;
            }
        }
    }
}
```

### Sorting an Array of Strings

- **Data types are different**
  - Array elements are `char*`
  - Swap variable is `char*`
- **Comparison operator is different**
  - The greater-than ("\( > \)") sign does not work
  - Need to use `strcmp()` function instead

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;the&quot;</td>
<td>&quot;quick&quot;</td>
<td>&quot;brown&quot;</td>
<td>&quot;fox&quot;</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;brown&quot;</td>
<td>&quot;fox&quot;</td>
<td>&quot;quick&quot;</td>
<td>&quot;the&quot;</td>
</tr>
</tbody>
</table>
String Sorting Function

```c
void sort(char *v[], int n)
{
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i+1; j < n; j++) {
            if (strcmp(v[i], v[j]) > 0) {
                char* swap = v[i];
                v[i] = v[j];
                v[j] = swap;
            }
        }
    }
}
```

Creating a Generic Function

- **Generic function**
  - A single `sort()` function that works for all data types
- **C's notion of data types is getting in our way**
  - We need to accept parameters in any type
    - `sort(int *v, int n)` is only good for integer arrays
    - `sort(char *v[], int n)` is only good for string arrays
  - We need to have local variables of any type
    - `int swap` is only good for swapping integers
    - `char* swap` is only good for swapping strings
- **Different types need different comparison operators**
  - Greater-than sign (">") is only good for numerical types
  - `strcmp()` is only good for strings
  - We need to be able to tell `sort()` what comparison function to use

Generalizing: Void Pointers

- **Generic pointers are the same as any other pointer**
  - Except they point to a variable with no specific type
  - Example: `void *datap = “CS217”;`
- **Difference:**
  - Regular pointers: compilers “know” what they point to
  - Void pointers: compilers “don’t know” what they point to
- **Common Uses:**
  - Abstract data types supporting polymorphism*
  - Pass pointer to function that could be any of several types

* Allowing the same definitions to be used with different types of data
Void Pointers in Sort

• Function parameters
  o Input: array of pointers to some unknown type

```
void sort(void *v[], int n)
```

• Local swap variable
  o Pointer to some unknown type

```
void *swap = v[i];
v[i] = v[j];
v[j] = swap;
```

• But, what about the comparison step?
  o Need to be able to pass a function to sort

Casting: Explicit Type Conversions

• Casting
  o As if the expression were assigned to a variable of the specified type
  o E.g., int *intp1 cast into void pointer by (void *) intp1

• C does many implicit conversions
  o E.g., function double sqrt(double)
    – Can be called as sqrt(2);
    – Which is treated as sqrt((double) 2);

• Sometimes useful to make conversion explicit
  o Documentation: making implicit type conversions explicit
    – E.g., getting the integer part of a floating-point number
    – Done by int_part = (int) float_number;
  o Control: overrule the compile by forcing conversions we want
    – E.g., getting the fractional part of a floating-point number
    – Done by frac_part = f - (int) f;

Generic Sort Function

```
void sort(void *v[], int n,
          int (*compare)(void *data p1, void *datap2))
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = i+1; j < n; j++) {
            if ((*compare)(v[i], v[j]) > 0) {
                void *swap = v[i];
                v[i] = v[j];
                v[j] = swap;
            }
        }
    }
}
```

`compare` is a pointer to a function that has two void* arguments and returns an int, and (*compare) is the function.
Using Generic Sort With String

```c
#include <stdio.h>
#include <string.h>
#include "sort.h"

int main(void) {
    char* w[4] = {"the", "quick", "brown", "fox"};
    sort((void **) w, 4, (int (*)(void*,void*)) strcmp);
    ...
}
```

Using Generic Sort With Integers

```c
#include <stdio.h>
#include "sort.h"

int CompareInts(void *datap1, void *datap2) {
    int *intp1 = (int *) datap1;
    int *intp2 = (int *) datap2;
    return (*intp1 - *intp2);
}

int main(void) {
    int* w[4];
    w[0] = malloc(sizeof(int));
    *(w[0]) = 7;
    ...
    sort((void **) w, 4, (int (*)(void*,void*))CompareInts);
    ...
}
```

Making “Array” an ADT

• Arrays in C are error prone
  ◦ Access elements before the array starts (e.g., v[-1])
  ◦ Access elements past the end of array (e.g., v[n])
  ◦ Modify the variable that keeps track of size (e.g., n)

• Protect programmers with an array ADT
  ◦ Create and delete an array
  ◦ Get the current length
  ◦ Read an array element
  ◦ Append, replace, remove
  ◦ Sort
Array ADT: Interface

typedef struct Array *Array_T;

Array_T Array_new(void);
void Array_free(Array_T array);

int Array_getLength(Array_T array);
void *Array_getData(Array_T array, int index);

void Array_append(Array_T array, void *datap);
void Array_replace(Array_T array, int index, void *datap);
void Array_remove(Array_T array, int index);

void Array_sort(Array_T array, int (*compare)(void *datap1, void *datap2));

array.h

Client Using Array ADT: Strings

#include "array.h"
#include <stdio.h>

int main(void) {
    Array_T array;
    int i;
    array = Array_new();
    Array_append(array, (void *) "COS217");
    Array_append(array, (void *) "IS");
    Array_append(array, (void *) "FUN");
    for (i = 0; i < Array_getLength(array); i++) {
        char *str = (char *) Array_getData(array, i);
        printf(str);
    }
    Array_free(array);
    return 0;
}

Client Using Array ADT: Integers

#include "array.h"
#include <stdio.h>

int main(void) {
    Array_T array;
    int one=1, two=2, three=3;
    int i;
    array = Array_new();
    Array_append(array, (void *) &one);
    Array_append(array, (void *) &two);
    Array_append(array, (void *) &three);
    for (i = 0; i < Array_getLength(array); i++) {
        int *datap = (int *) Array_getData(array, i);
        printf("%d ", *datap);
    }
    Array_free(array);
    return 0;
}
# Array ADT Implementation

```c
#include "array.h"

enum {MAX_ELEMENTS = 128};

struct Array {
    void *elements[MAX_ELEMENTS];
    int num_elements;
};

Array_T Array_new(void) {
    Array_T array = malloc(sizeof(struct Array));
    array->num_elements = 0;
    return array;
}

void Array_free(Array_T array) {
    free(array);
}

int Array_getLength(Array_T array) {
    return array->num_elements;
}

void *Array_getData(Array_T array, int index) {
    return array->elements[index];
}

void Array_append(Array_T array, void *datap) {
    int index = array->num_elements;
    array->elements[index] = datap;
    array->num_elements++;
}

void Array_replace(Array_T array, int index, void *datap) {
    array->elements[index] = datap;
}

void Array_insert(Array_T array, int index, void *datap) {
    int i;
    /* Shift elements to the right to make room for new entry */
    for (i = array->num_elements; i > index; i--)
        array->elements[i] = array->elements[i-1];
    /* Add the new element in the now-free location */
    array->elements[index] = datap;
    array->num_elements++;
}

void Array_remove(Array_T array, int index) {
    int i;
    /* Shift elements to the left to overwrite freed spot */
    for (i = index+1; i < array->num_elements; i++)
        array->elements[i-1] = array->elements[i];
    array->num_elements--;
}
```
void Array_sort(Array_T array, 
    int (*compare)(void *datap1, void *datap2)) 
{
    int i, j;
    for (i = 0; i < array->num_elements; i++) {
        for (j = i+1; j < array->num_elements; j++) {
            if (*compare(array->elements[i], array->elements[j]) > 0) {
                void *swap = array->elements[i];
                array->elements[i] = array->elements[j];
                array->elements[j] = swap;
            }
        }
    }
}

Stupid Programmer Tricks

• qsort takes int (*compare)(const void *, const void *)
  ○ Comparison function returns integer greater than, equal, less than zero if first argument is greater than, equal, less than second

• Common approach:
  int ItemCompare(const void *pA, const void *pB) {
    Item *a = pA, *b = pB;
    return(a->field - b->field);
  }

• Bad idea when field is float or “long long” (64 bit)

Summary

• Module supporting operations on single data structure
  ○ Interface declares operations, not data structure
  ○ Interface provides access to simple, complete set of operations
  ○ Interface provides flexibility and extensibility

• Trick is providing functionality AND generality
  ○ Take advantage of features of programming language
    – void pointers
    – function pointers

• Advantages
  ○ Provide complete set of commonly used functions (re-use)
  ○ Implementation is hidden from client (encapsulation)
  ○ Can use for multiple types (polymorphism)