Abstract Data Types

CS 217

Recap of Last Time

- Modularity is key to good software
  - Decompose program into modules
  - Provide clear and flexible interfaces
- Advantages
  - Separate compilation
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes

Modularity in C

- Use features of programming language to enhance modularity
  - **`struct`**
  - **`typedef`**
  - opaque pointers
  - void pointers
  - function pointers

Structures

stringarray.h

```
#define MAX_STRINGS 128

struct StringArray {
  char *strings[MAX_STRINGS];
  int nstrings;
};

extern void ReadStrings(struct StringArray *stringarray, FILE *fp);
extern void WriteStrings(struct StringArray *stringarray, FILE *fp);
extern void SortStrings(struct StringArray *stringarray);
```

sort.c

```
#include <stdio.h>
#include "stringarray.h"

int main()
{
  struct StringArray *stringarray = malloc( sizeof(struct StringArray) );
  stringarray->nstrings = 0;

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray);
  WriteStrings(stringarray, stdout);

  free(stringarray);
  return 0;
}
```

Typedef

stringarray.h

```
#define MAX_STRINGS 128

typedef struct StringArray {
  char *strings[MAX_STRINGS];
  int nstrings;
} *StringArray_T;

extern void ReadStrings(StringArray_T stringarray, FILE *fp);
extern void WriteStrings(StringArray_T stringarray, FILE *fp);
extern void SortStrings(StringArray_T stringarray);
```

sort.c

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  StringArray_T stringarray = malloc( sizeof(struct StringArray) );
  stringarray->nstrings = 0;

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray);
  WriteStrings(stringarray, stdout);

  free(stringarray);
  return 0;
}
```

Opaque Pointers

stringarray.h

```
typedef struct StringArray *StringArray_T;

extern StringArray_T NewStrings(void);
extern void FreeStrings(StringArray_T stringarray);

extern void ReadStrings(StringArray_T stringarray, FILE *fp);
extern void WriteStrings(StringArray_T stringarray, FILE *fp);
extern void SortStrings(StringArray_T stringarray);
```

stringarray.c

```
#include “stringarray.h”

#define MAX_ELEMENTS 128

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

etc.
```

Abstract Data Types (ADTs)

- 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

stringarray.h

```
typedef struct StringArray *StringArray_T;

extern StringArray_T NewStrings(void);
extern void FreeStrings(StringArray_T stringarray);

extern void ReadStrings(StringArray_T stringarray, FILE *fp);
extern void WriteStrings(StringArray_T stringarray, FILE *fp);
extern void SortStrings(StringArray_T stringarray);
```

What would be a more general ADT?

Array ADT - Interface

array.h

```
typedef struct Array *Array_T;

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

extern int Array_getLength(Array_T array);
extern char *Array_getData(Array_T array, int index);
extern int Array_getIndex(Array_T array, char *str);

extern void Array_append(Array_T array, char *str);
extern void Array_insert(Array_T array, int index, char *str);
extern void Array_replace(Array_T array, int index, char *str);
extern void Array_remove(Array_T array, int index);
extern void Array_sort(Array_T array);
```

Example ADT - Client

client.c

```
#include “array.h”
#include <stdio.h>

int main()
{
  Array_T array;
  int i;

  array = Array_new();

  Array_append(array, “CS217”);
  Array_append(array, “IS”);
  Array_append(array, “FUN”);

  for (i = 0; i < Array_getLength(array); i++) {
    char *str = Array_getData(array, i);
    printf(str);
  }

  Array_free(array);

  return 0;
}
```

Example ADT - Implementation

array.c (1 of 5)

```
#include “array.h”

#define MAX_ELEMENTS 128

struct Array {
  char *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);
}
```

Example ADT - Implementation

array.c (2 of 5)

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


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


int Array_getIndex(Array_T array, char *str)
{
  int index;
  for (index = 0; index < array->num_elements; index++) {
    if (array->elements[index] == str) {
      return index;
      break;
    }
  }
  return -1;
}
```

Example ADT - Implementation

array.c (3 of 5)

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


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

Example ADT - Implementation

array.c (4 of 5)

```
void Array_insert(Array_T array, int index, char *str)
{
  int i;

  for (i = array->num_elements; i > index; i--)
    array->elements[i] = array->elements[i-1];

  array->elements[index] = str;
  array->num_elements++;
}


void Array_remove(Array_T array, int index)
{
  int i;

  for (i = index+1; i < array->num_elements; i++)
    array->elements[i-1] = array->elements[i];

  array->num_elements--;
}
```

Example ADT - Implementation

array.c (5 of 5)

```
void Array_sort(Array_T array)
{
  int i, j;

  for (i = 0; i < array->num_elements; i++) {
    for (j = i+1; j < array->num_elements; j++) {
      if (strcmp(array->elements[i], array->elements[j]) > 0) {
        char *swap = array->elements[i];
        array->elements[i] = array->elements[j];
        array->elements[j] = swap;
      }
    }
  }
}
```

Abstract Data Types (ADTs)

- 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

array.h

```
typedef struct Array *Array_T;

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

extern int Array_getLength(Array_T array);
extern char *Array_getData(Array_T array, int index);
extern int Array_getIndex(Array_T array, char *str);

extern void Array_append(Array_T array, char *str);
extern void Array_insert(Array_T array, int index, char *str);
extern void Array_replace(Array_T array, int index, char *str);
extern void Array_remove(Array_T array, int index);
extern void Array_sort(Array_T array);
```

Abstract Data Types (ADTs)

- 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

array.h

```
typedef struct Array *Array_T;

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

extern int Array_getLength(Array_T array);
extern char *Array_getData(Array_T array, int index);
extern int Array_getIndex(Array_T array, char *str);

extern void Array_append(Array_T array, char *str);
extern void Array_insert(Array_T array, int index, char *str);
extern void Array_replace(Array_T array, int index, char *str);
extern void Array_remove(Array_T array, int index);
extern void Array_sort(Array_T array);
```

But this array ADT still only holds strings

Polymorphism - Void Pointers

array.h

```
typedef struct Array *Array_T;

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

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

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

Example ADT - Client 1

string_client.c

```
#include “array.h”
#include <stdio.h>

int main()
{
  Array_T array;
  int i;

  array = Array_new();

  Array_append(array, (void *) “CS217”);
  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;
}
```

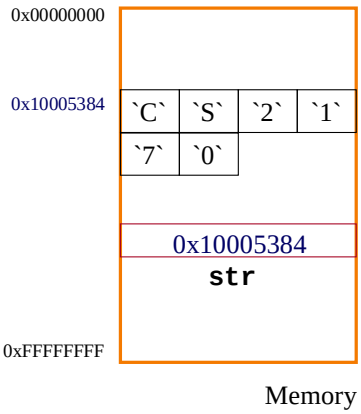
Example ADT - Client 2

int_client.c

```
#include “array.h”
#include <stdio.h>

int main()
{
  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;
}
```

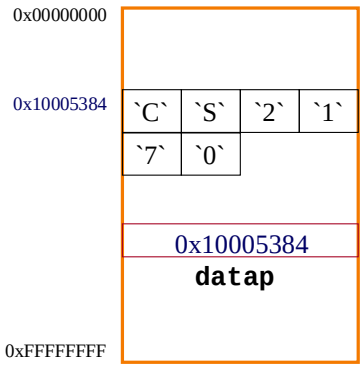
Pointers

- Pointers are variables whose value is an address
  - They usually point to a variable of a specific type
  - Example: **`char *str = “CS217”;`**



Void Pointers

- Void pointers are the same as any other pointer
  - Except they point to a variable with no specific type
  - Example: **`void *datap = "CS217";`**
- Difference:
  - Compiler does not check type when dereference
- Common Uses:
  - Abstract data types supporting polymorphism
  - Pass pointer to function that could be any of several types



Void Pointer Implementation

array.c (1 of 4)

```
#include "array.h"

#define 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);
}
```

SAME AS BEFORE

EXCEPT ...

Example ADT - Implementation

array.c

```
void Array_sort(Array_T array)
{
  int i, j;

  for (i = 0; i < array->num_elements; i++) {
    for (j = i+1; j < array->num_elements; j++) {
      if (strcmp(array->elements[i], array->elements[j]) > 0) {
        char *swap = array->elements[i];
        array->elements[i] = array->elements[j];
        array->elements[j] = swap;
      }
    }
  }
}
```

How do we compare two data elements
if we don't know their types?

Function Pointers - Interface

array.h

```
typedef struct Array *Array_T;

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

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

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

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

Pass function pointer to ADT

Function Pointers - Implementation

array.c

```
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) {
        char *swap = array->elements[i];
        array->elements[i] = array->elements[j];
        array->elements[j] = swap;
      }
    }
  }
}
```

Call function by dereferencing function pointer

Function Pointers - Client

int_client.c

```
#include “array.h”
#include <stdio.h>

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

int main()
{
  Array_T array;
  int one=1, two=2, three=3;

  array = Array_new();
  Array_append(array, (void *) &one);
  Array_append(array, (void *) &two);
  Array_append(array, (void *) &three);
  Array_sort(array, CompareInts);

  etc.
}
```

Every function name (e.g., CompareInts) represents the address of a function (i.e., a pointer to a function).

Summary of Abstract Data Types

- 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)