Abstract Data Types

CS 217

Recap

- Good software
  - Decompose program into modules
  - Provide clear separations between module implementations
    - Design clean interfaces
    - Use scope to ensure separation
    - Document interfaces clearly
- Advantages
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes
  - Separate compilation

Example Program 1

```
#include <stdio.h>
#include <string.h>

int main()
{
  char *strings[128];
  char string[256];
  char *p1, *p2;
  int nstrings;
  int found;
  int i, j;

  nstrings = 0;
  while (fgets(string, 256, stdin)) {
    for (i = 0; i < nstrings; i++) {
      found = 1;
      for (p1 = string, p2 = strings[i]; *p1 && *p2; p1++, p2++) {
        if (*p1 > *p2) {
          found = 0;
          break;
        }
      }
      if (found) break;
    }
    for (j = nstrings; j > i; j--)
      strings[j] = strings[j-1];
    strings[i] = strdup(string);
    nstrings++;
    if (nstrings >= 128) break;
  }
  for (i = 0; i < nstrings; i++)
    fprintf(stdout, "%s", strings[i]);

  return 0;
}
```

Example Program 2

```
#include <stdio.h>
#include <string.h>


#define MAX_STRINGS 128
#define MAX_STRING_LENGTH 256


void ReadStrings(char **strings, int *nstrings, int maxstrings, FILE *fp)
{
  char string[MAX_STRING_LENGTH];

  *nstrings = 0;
  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    strings[(*nstrings)++] = strdup(string);
    if (*nstrings >= maxstrings) break;
  }
}


void WriteStrings(char **strings, int nstrings, FILE *fp)
{
  int i;

  for (i = 0; i < nstrings; i++)
    fprintf(fp, "%s", strings[i]);
}


int CompareStrings(char *string1, char *string2)
{
  char *p1 = string1;
  char *p2 = string2;

  while (*p1 && *p2) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;
    p1++;
    p2++;
  }

  return 0;
}
```

```
void SortStrings(char **strings, int nstrings)
{
  int i, j;

  for (i = 0; i < nstrings; i++) {
    for (j = i+1; j < nstrings; j++) {
      if (CompareStrings(strings[i], strings[j]) > 0) {
        char *swap = strings[i];
        strings[i] = strings[j];
        strings[j] = swap;
      }
    }
  }
}


int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

Separate Compilation

```
#include <stdio.h>
#include <string.h>


#define MAX_STRINGS 128
#define MAX_STRING_LENGTH 256


void ReadStrings(char **strings, int *nstrings, int maxstrings, FILE *fp)
{
  char string[MAX_STRING_LENGTH];

  *nstrings = 0;
  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    strings[(*nstrings)++] = strdup(string);
    if (*nstrings >= maxstrings) break;
  }
}


void WriteStrings(char **strings, int nstrings, FILE *fp)
{
  int i;

  for (i = 0; i < nstrings; i++)
    fprintf(fp, "%s", strings[i]);
}


int CompareStrings(char *string1, char *string2)
{
  char *p1 = string1;
  char *p2 = string2;

  while (*p1 && *p2) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;
    p1++;
    p2++;
  }

  return 0;
}
```

```
void SortStrings(char **strings, int nstrings)
{
  int i, j;

  for (i = 0; i < nstrings; i++) {
    for (j = i+1; j < nstrings; j++) {
      if (CompareStrings(strings[i], strings[j]) > 0) {
        char *swap = strings[i];
        strings[i] = strings[j];
        strings[j] = swap;
      }
    }
  }
}
```

```
int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

in separate file

Separate Compilation

stringarray.h

```
extern void ReadStrings(char **strings, int *nstrings, int maxstrings, FILE *fp);
extern void WriteStrings(char **strings, int nstrings, FILE *fp);
extern void SortStrings(char **strings, int nstrings);
```

sort.c

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

#define MAX_STRINGS 128

int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

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);
```

sort.c

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

int main()
{
  StringArray_T stringarray = NewStrings();

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

  FreeStrings(stringarray);

  return 0;
}
```

Abstract Data Type

stringarray.h

```
typedef struct StringArray *StringArray_T;

extern StringArray_T StringArray_new(void);
extern void StringArray_free(StringArray_T stringarray);

extern void StringArray_read(StringArray_T stringarray, FILE *fp);
extern void StringArray_write(StringArray_T stringarray, FILE *fp);
extern void StringArray_sort(StringArray_T stringarray);
```

sort.c

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

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

Abstract Data Types

- Module supporting operations on single data type
  - Interface declares operations, not data structure
  - Implementation is hidden from client (encapsulation)
- Common practice
  - Use scope rules to ensure encapsulation (e.g., opaque pointers)
  - Allocation and deallocation of data structure handled by module
  - Names of functions and variables begin with <modulename>_
- Advantages
  - Provides separation of code in different modules
  - Localizes effect of each change to a single module
  - Easier to modify, test, and debug

Implementation

stringarray.c (1 of 5)

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


#define MAX_STRINGS 128
#define MAX_STRING_LENGTH 256


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


(continued on next slide)
```

Implementation

stringarray.c (2 of 5)

```
StringArray_T StringArray_new(void)
{
  StringArray_T s = malloc(sizeof(struct StringArray));
  s->nstrings = 0;
  return s;
}


void StringArray_free(StringArray_T stringarray)
{
  free(stringarray);
}
```

(continued on next slide)

Implementation

stringarray.c (3 of 5)

```
void StringArray_write(StringArray_T s, FILE *fp)
{
  int i;

  for (i = 0; i < s->nstrings; i++)
    fprintf(fp, "%s", s->strings[i]);
}
```

(continued on next slide)

Implementation

stringarray.c (4 of 5)

```
void StringArray_read(StringArray_T s, FILE *fp)
{
  char string[MAX_STRING_LENGTH];

  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    s->strings[(s->nstrings)++] = strdup(string);
    if (s->nstrings >= MAX_STRINGS) break;
  }
}
```

(continued on next slide)

Implementation

stringarray.c (5 of 5)

```
static int StringArray_compare(char *string1, char *string2)
{
  return strcmp(string1, string2);
}


void StringArray_sort(StringArray_T s)
{
  int i, j;

  for (i = 0; i < s->nstrings; i++) {
    for (j = i+1; j < s->nstrings; j++) {
      if (StringArray_compare(s->strings[i], s->strings[j]) > 0) {
        char *swap = s->strings[i];
        s->strings[i] = s->strings[j];
        s->strings[j] = swap;
      }
    }
  }
}
```

Function Pointers

stringarray.h

```
typedef struct StringArray *StringArray_T;

extern StringArray_T StringArray_new(void);
extern void StringArray_free(StringArray_T stringarray);

extern void StringArray_read(StringArray_T stringarray, FILE *fp);
extern void StringArray_write(StringArray_T stringarray, FILE *fp);
extern void StringArray_sort(StringArray_T stringarray,
                             int (*compare)(const char *s1, const char *s2));
```

sort.c

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

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, strcmp);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

Function Pointers

stringarray.c

```
void StringArray_sort(StringArray_T s, int (*compare)(const char *s1, const char *s2));
)
{
  int i, j;

  for (i = 0; i < s->nstrings; i++) {
    for (j = i+1; j < s->nstrings; j++) {
      if ((*compare)(s->strings[i], s->strings[j]) > 0) {
        char *swap = s->strings[i];
        s->strings[i] = s->strings[j];
        s->strings[j] = swap;
      }
    }
  }
}
```

Dynamic Allocation

stringarray.c

```
...

#define MAX_STRINGS 128

...

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

...

void StringArray_read(StringArray_T s, FILE *fp)
{
  char string[MAX_STRING_LENGTH];

  s->nstrings = 0;
  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    s->strings[(s->nstrings)++] = strdup(string);
    if (s->nstrings >= MAX_STRINGS) break;
  }
}

...
```

Dynamic Allocation

stringarray.c

```
...

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

...

void StringArray_read(StringArray_T s, FILE *fp)
{
  char string[MAX_STRING_LENGTH];

  s->nstrings = 0;
  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    StringArray_grow(nstrings+1);
    s->strings[(s->nstrings)++] = strdup(string);
  }
}

...
```

Summary

- Abstract Data Types
  - Modules supporting operations on data type
  - Well-designed interfaces hide implementations, but provide flexibility
  - ADTs facilitate modifications, debugging, testing, etc.

sort.c

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

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, strcmp);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```