Modules

CS 217

The C Programming Language

- Systems programming language
  - originally used to write Unix and Unix tools
  - data types and control structures close to most machines
  - now also a popular application programming language
- Notable features
  - all functions are call-by-value
  - pointer (address) arithmetic
  - simple scope structure
  - I/O and memory mgmt facilities provided by libraries
- History
  - BCPL à B à C à K&R C à ANSI C
    1960 1970 1972 1978 ↘ 1988
  - LISP à Smalltalk à C++ à Java

Example Program 1

```c
#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;
}
```

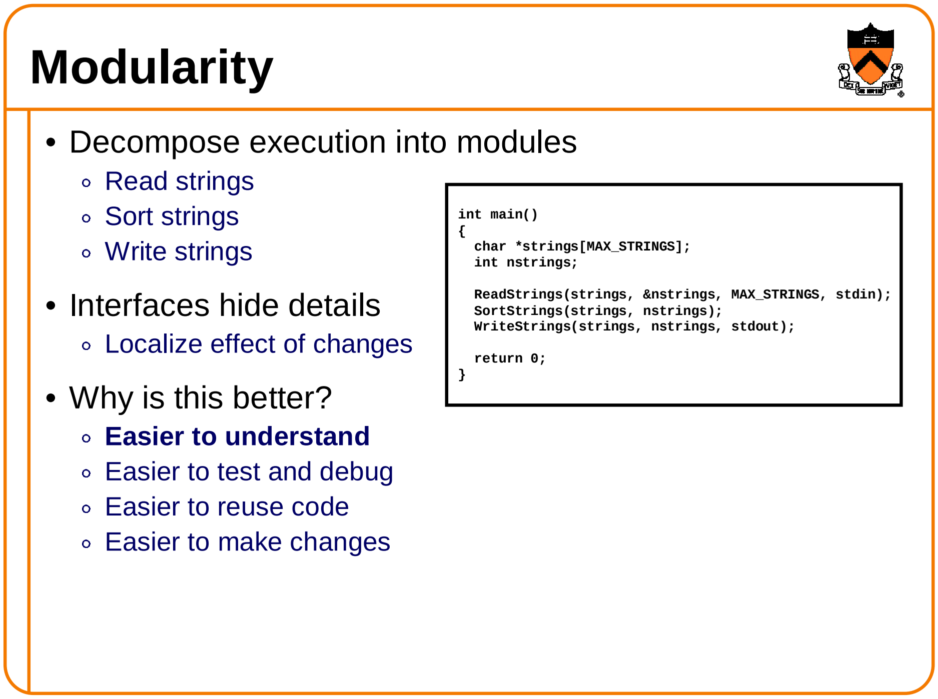
What does this program do?

Example Program 2

```c
#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;
}
```

```c
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;
}
```

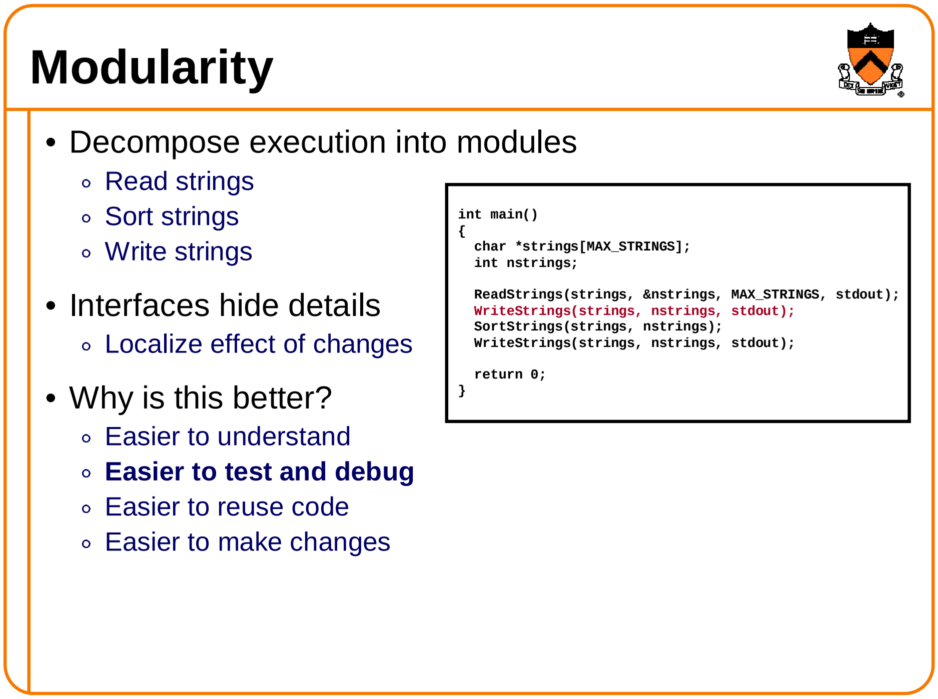
What does this program do?



Modularity

- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - **Easier to understand**
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes

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

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

  return 0;
}
```



Modularity

- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - **Easier to test and debug**
  - Easier to reuse code
  - Easier to make changes

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

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

  return 0;
}
```

Modularity

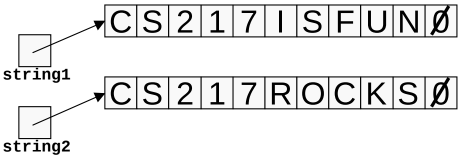
- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - Easier to test and debug
  - **Easier to reuse code**
  - Easier to make changes

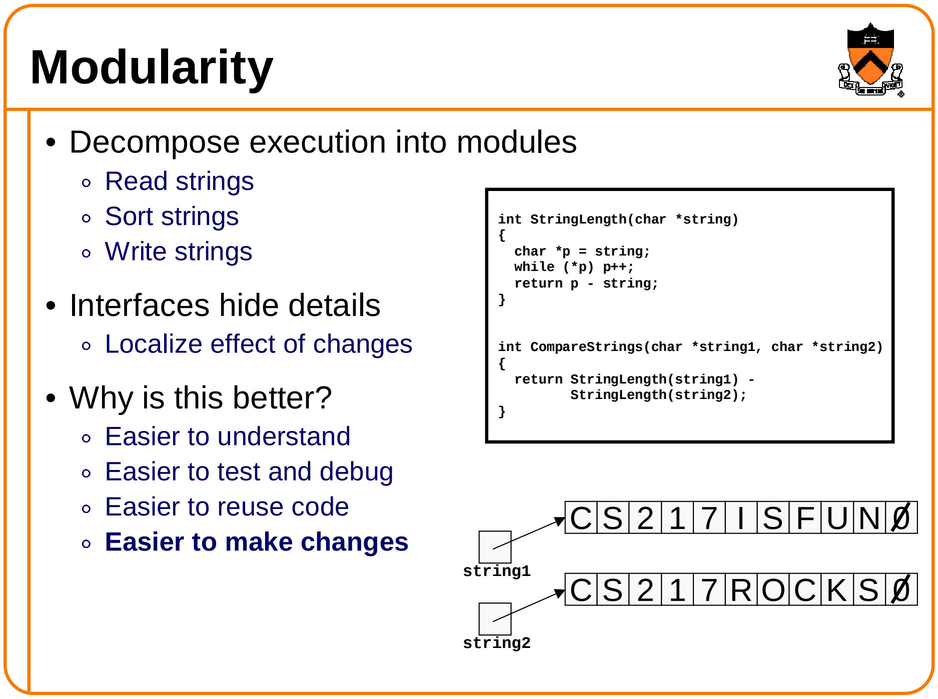
```
MergeFiles(FILE *fp1, FILE *fp2)
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, fp1);
  WriteStrings(strings, nstrings, stdout);

  ReadStrings(strings, &nstrings, MAX_STRINGS, fp2);
  WriteStrings(strings, nstrings, stdout);
}
```

Modularity

- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - **Easier to make changes**

```
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;
}
```



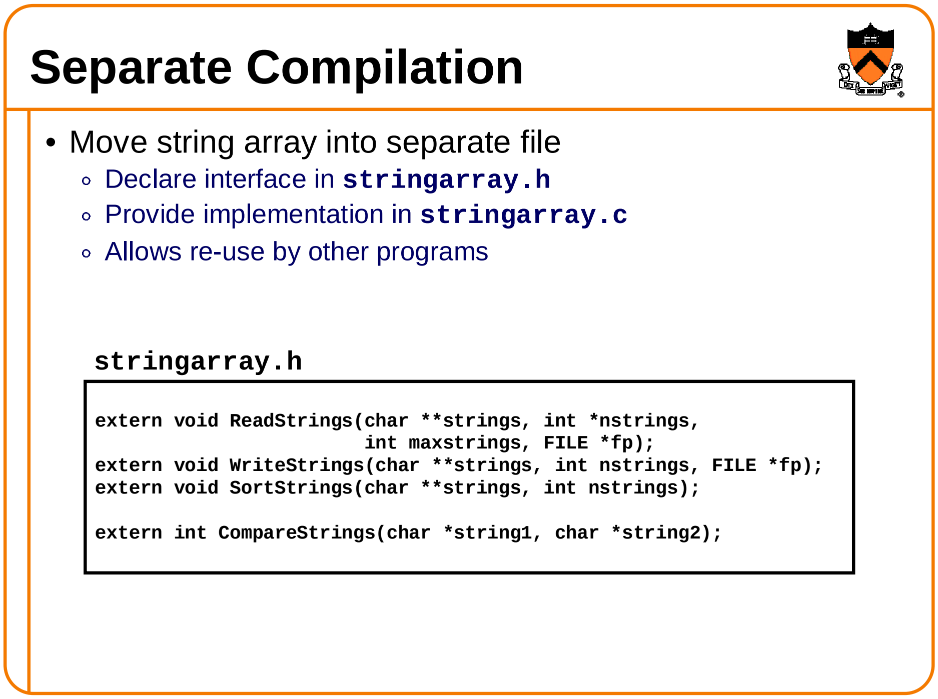


Modularity

- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - **Easier to make changes**

```
int StringLength(char *string)
{
  char *p = string;
  while (*p) p++;
  return p - string;
}


int CompareStrings(char *string1, char *string2)
{
  return StringLength(string1) -
         StringLength(string2);
}
```

string1	C	S	2	1	7	I	S	F	U	N	Ø
string2	C	S	2	1	7	R	O	C	K	S	Ø

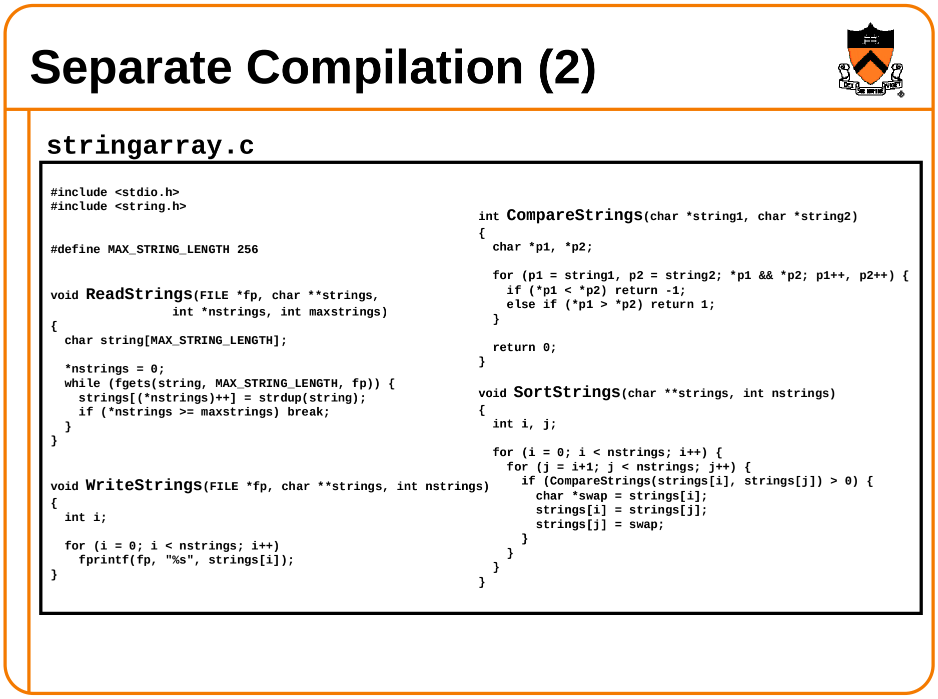


Separate Compilation

- Move string array into separate file
  - Declare interface in **`stringarray.h`**
  - Provide implementation in **`stringarray.c`**
  - Allows re-use by other programs

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);

extern int CompareStrings(char *string1, char *string2);
```

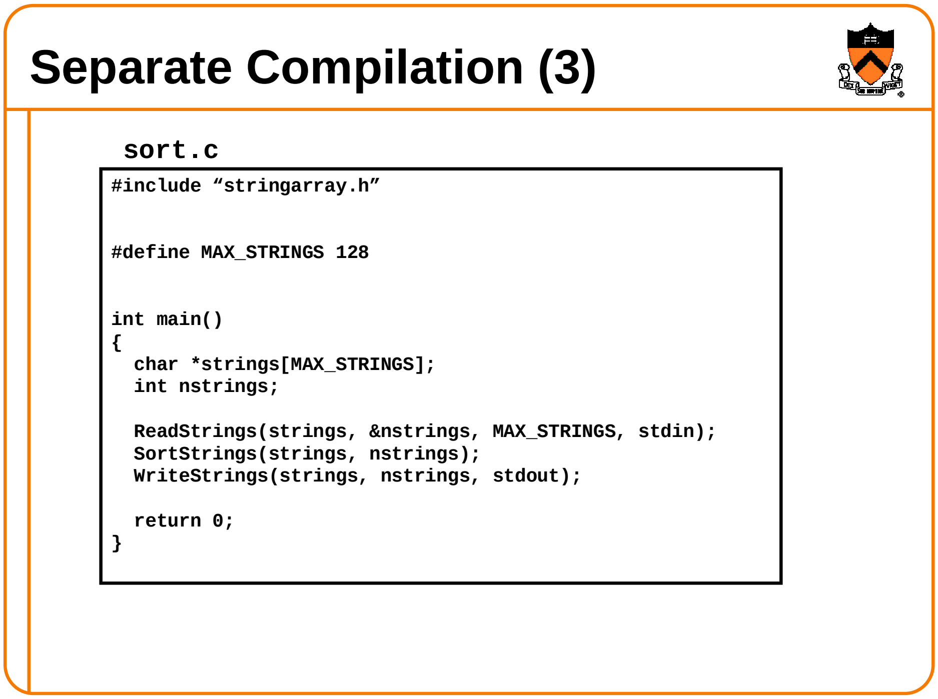


Separate Compilation (2)

stringarray.c

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


#define MAX_STRING_LENGTH 256


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

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


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

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

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

  for (p1 = string1, p2 = string2; *p1 && *p2; p1++, p2++) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;
  }

  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;
      }
    }
  }
}
```



Separate Compilation (3)

sort.c

```
#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;
}
```

Separate Compilation (4)

Makefile

```
sort: sort.o stringarray.a
        cc -o sort sort.o stringarray.a

sort.o: sort.c stringarray.h
        cc -c sort.c

stringarray.a: stringarray.c
        cc -c stringarray.c
        ar ur stringarray.a stringarray.o

clean:
        rm sort sort.o sortarray.a sortarray.o
```

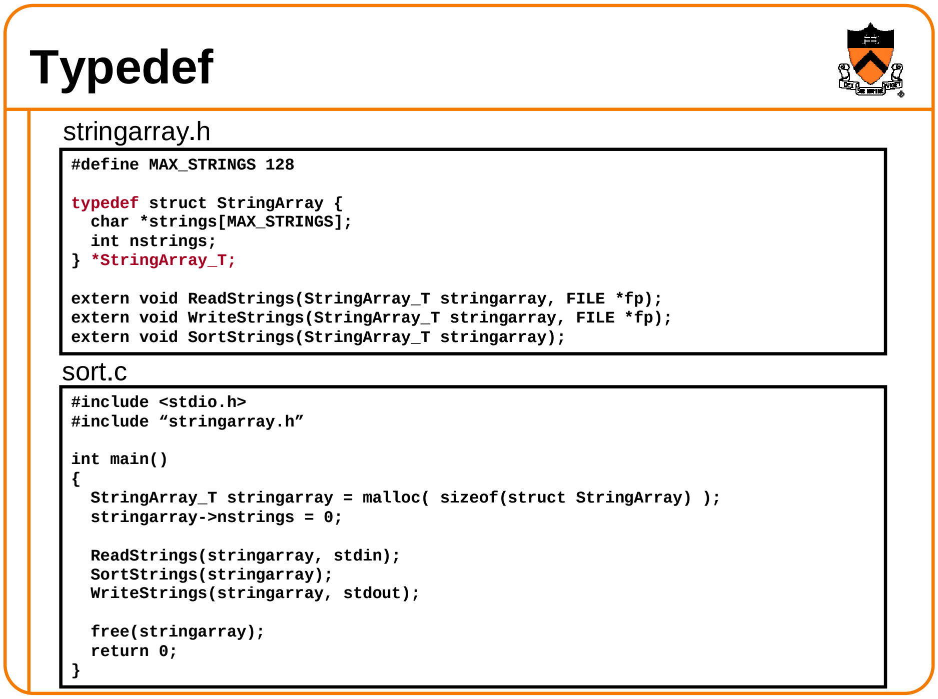
Structures

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;
}
```



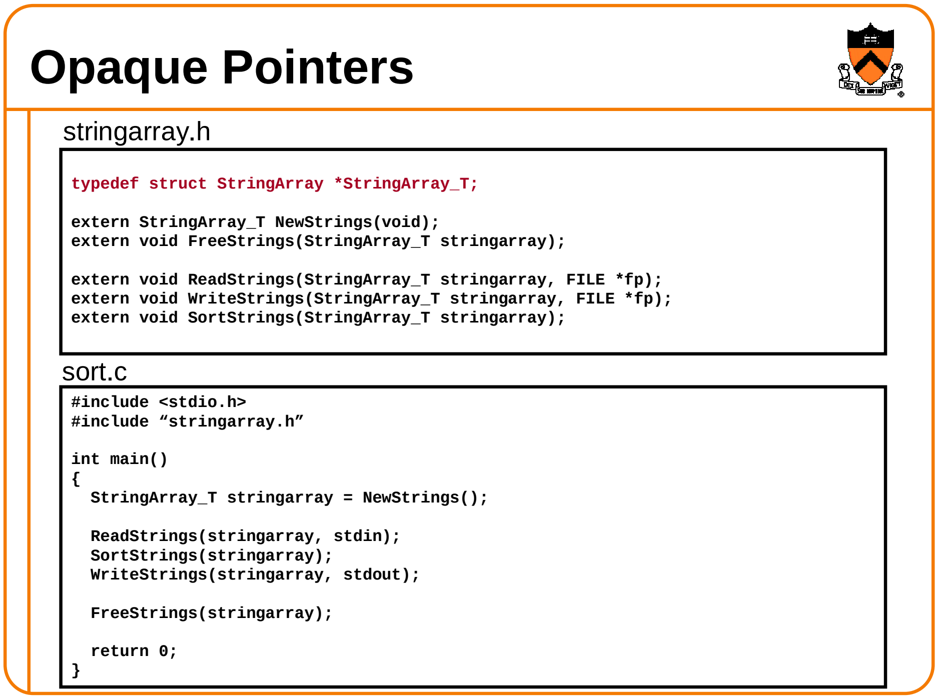
Typedef

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;
}
```

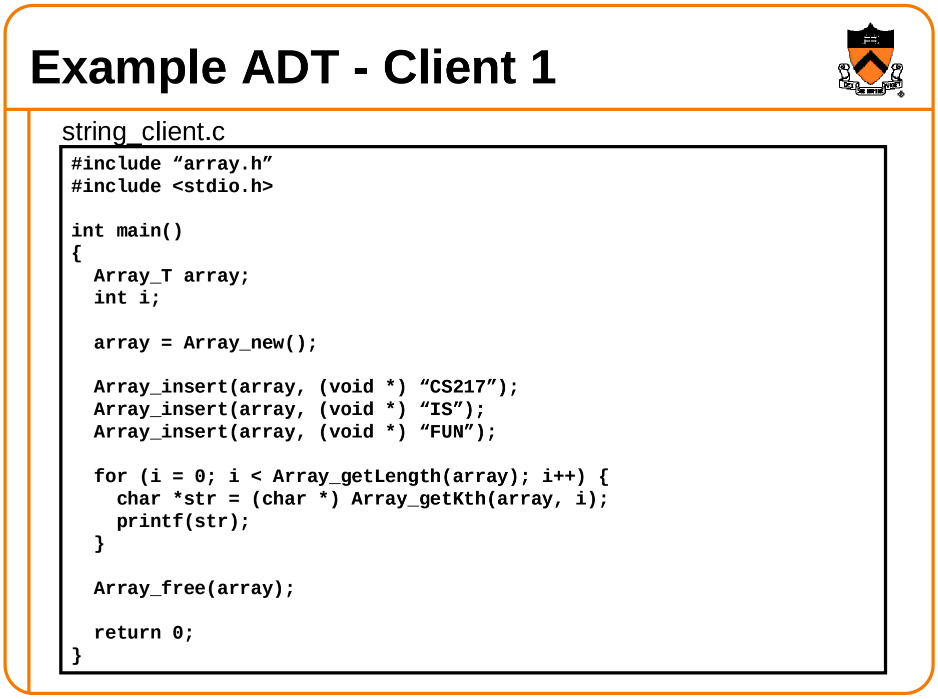
Abstract Data Types

- Module supporting operations on single data structure
  - Interface declares operations, not data structure
  - Implementation is hidden from client (encapsulation)
  - Use features of programming language to ensure encapsulation
- Common practice
  - Allocation and deallocation of data structure handled by module
  - Names of functions and variables begin with <modulename>_
  - Provide as much generality/flexibility in interface as possible
  - Use void pointers to allow polymorphism

Example ADT - Interface

array.h

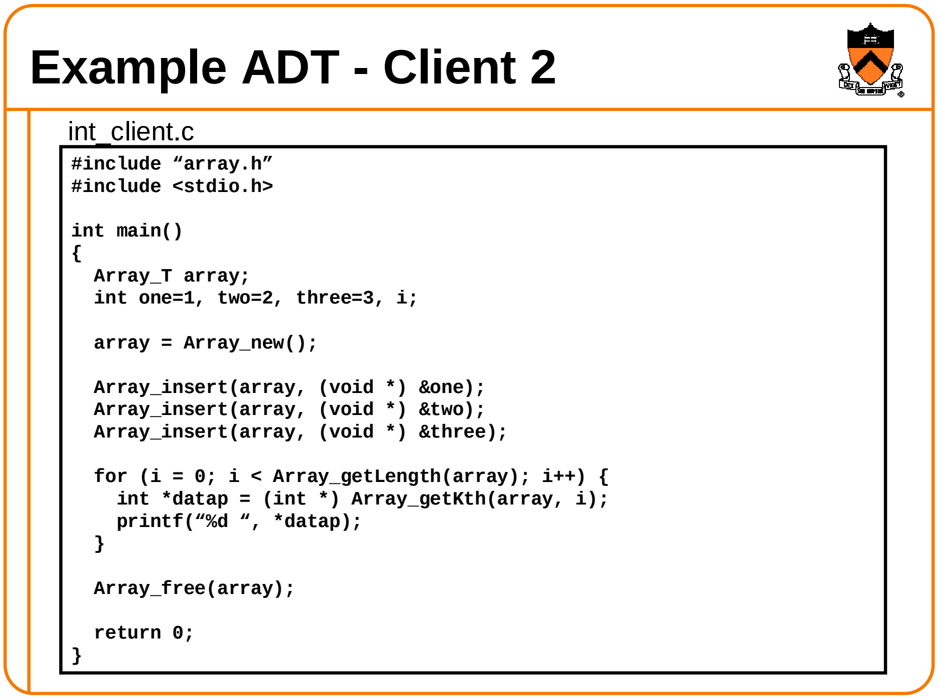
```
#ifndef ARRAY_H
#define ARRAY_H

typedef struct Array *Array_T;

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

extern void Array_insert(Array_T array, void *datap);
extern void Array_remove(Array_T array, void *datap);

extern int Array_getLength(Array_T array);
extern void *Array_getKth(Array_T array, int k);

#endif
```



Example ADT - Client 1

string_client.c

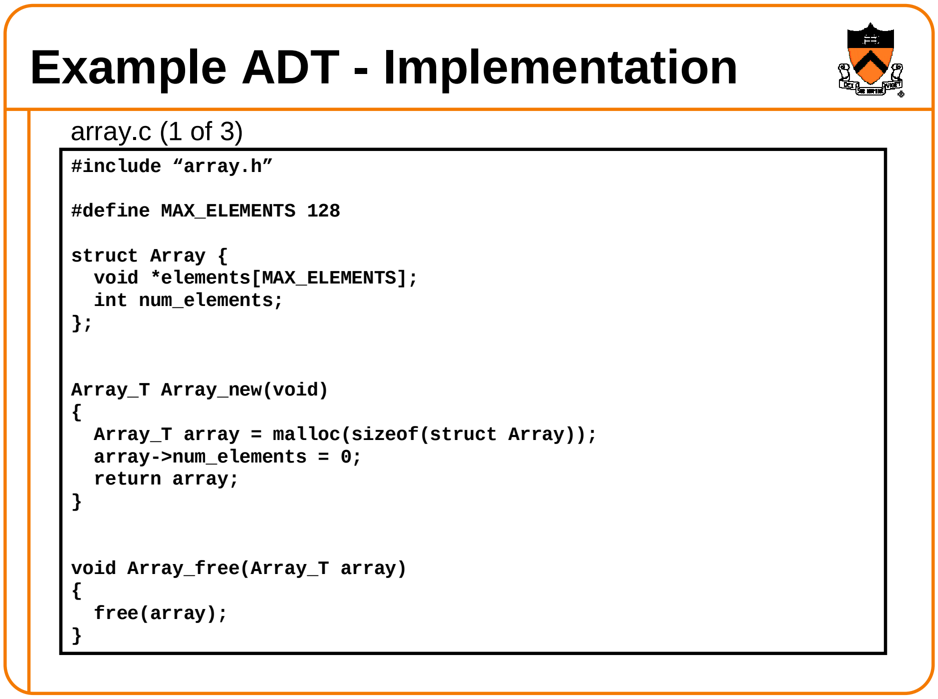
```
#include “array.h”
#include <stdio.h>

int main()
{
  Array_T array;
  int i;

  array = Array_new();

  Array_insert(array, (void *) “CS217”);
  Array_insert(array, (void *) “IS”);
  Array_insert(array, (void *) “FUN”);

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

  Array_free(array);

  return 0;
}
```



Example ADT - Client 2

int_client.c

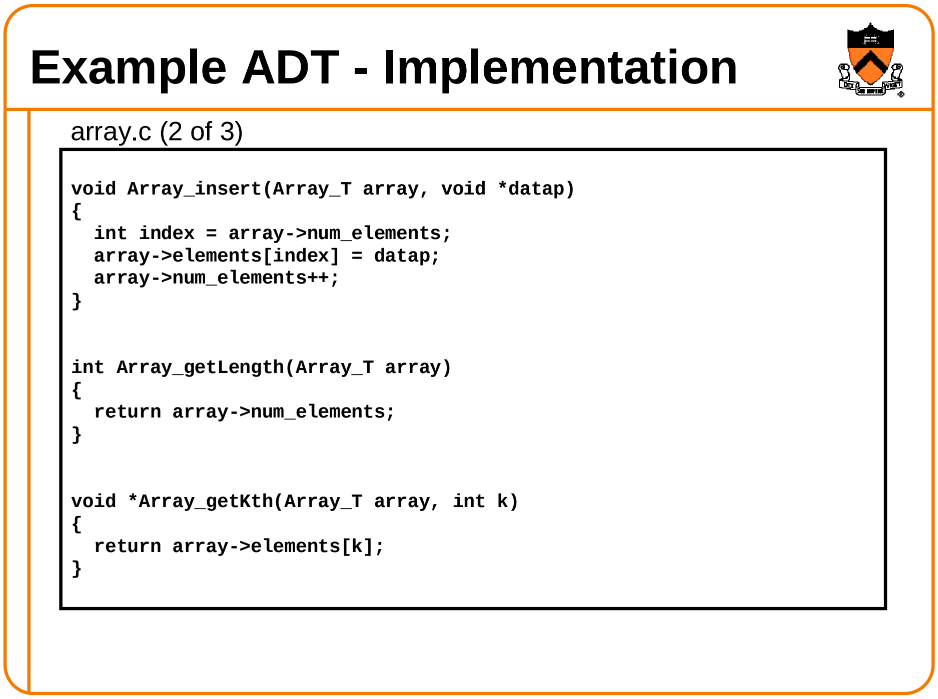
```
#include “array.h”
#include <stdio.h>

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

  array = Array_new();

  Array_insert(array, (void *) &one);
  Array_insert(array, (void *) &two);
  Array_insert(array, (void *) &three);

  for (i = 0; i < Array_getLength(array); i++) {
    int *datap = (int *) Array_getKth(array, i);
    printf(“%d “, *datap);
  }

  Array_free(array);

  return 0;
}
```



Example ADT - Implementation

array.c (1 of 3)

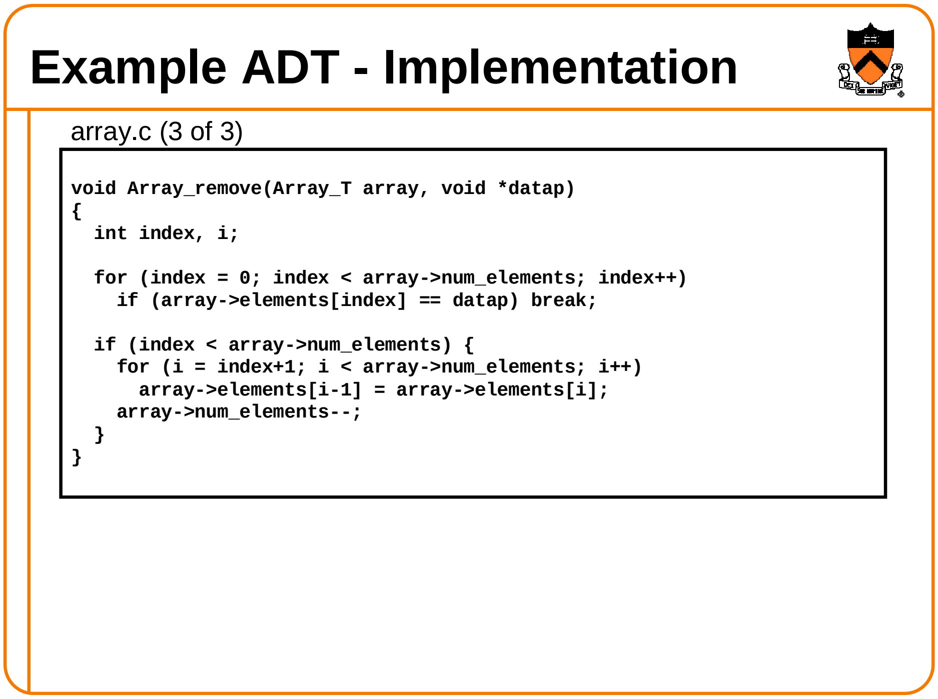
```
#include “array.h”

#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);
}
```



Example ADT - Implementation

array.c (2 of 3)

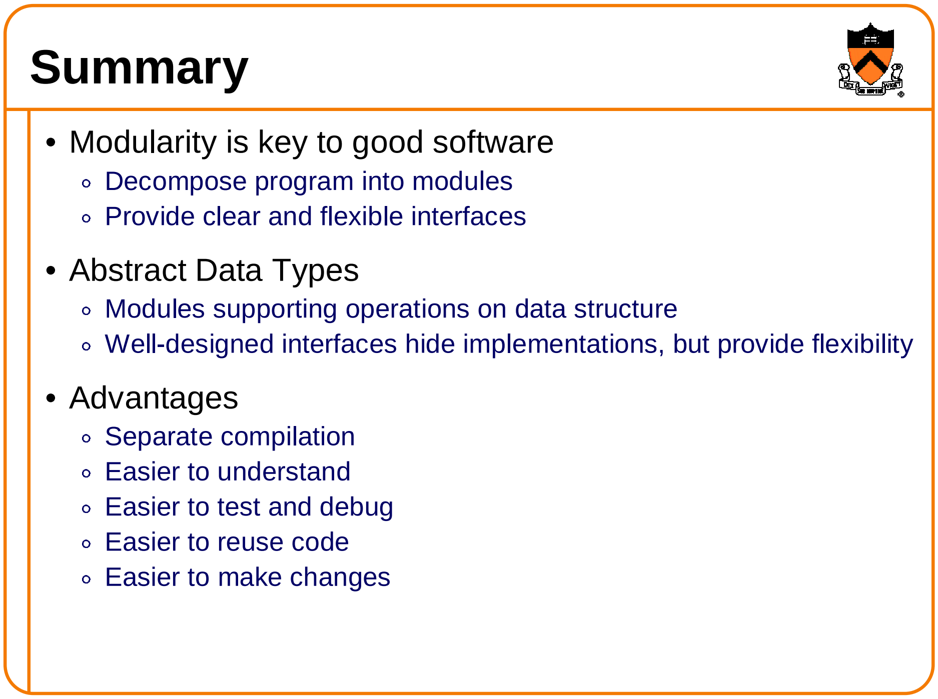
```

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


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


void *Array_getKth(Array_T array, int k)
{
  return array->elements[k];
}
```



Example ADT - Implementation

array.c (3 of 3)

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

  for (index = 0; index < array->num_elements; index++)
    if (array->elements[index] == datap) break;

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



Summary

- Modularity is key to good software
  - Decompose program into modules
  - Provide clear and flexible interfaces
- Abstract Data Types
  - Modules supporting operations on data structure
  - Well-designed interfaces hide implementations, but provide flexibility
- Advantages
  - Separate compilation
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes