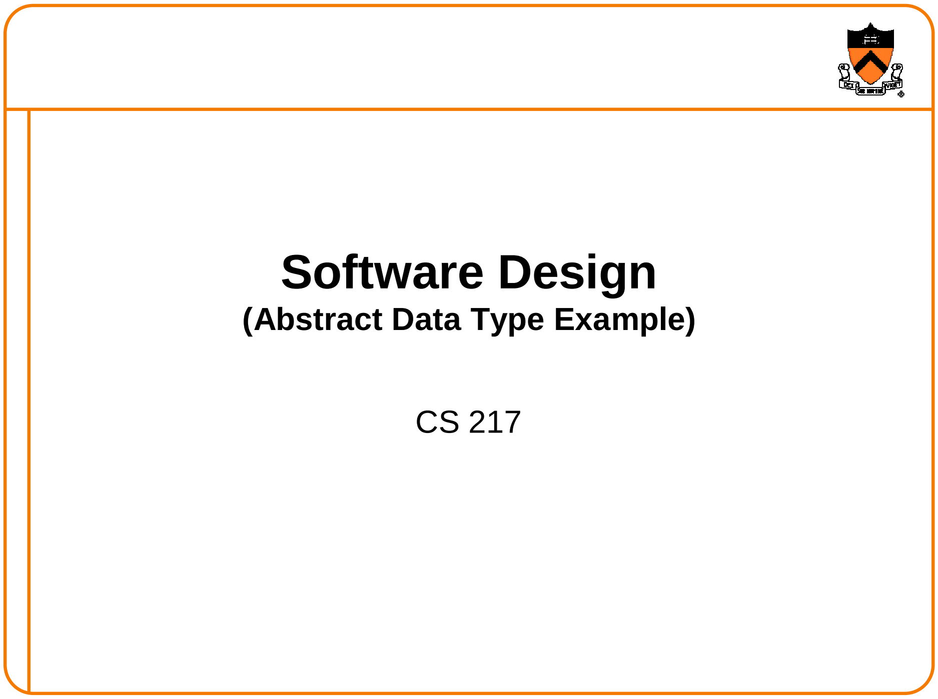
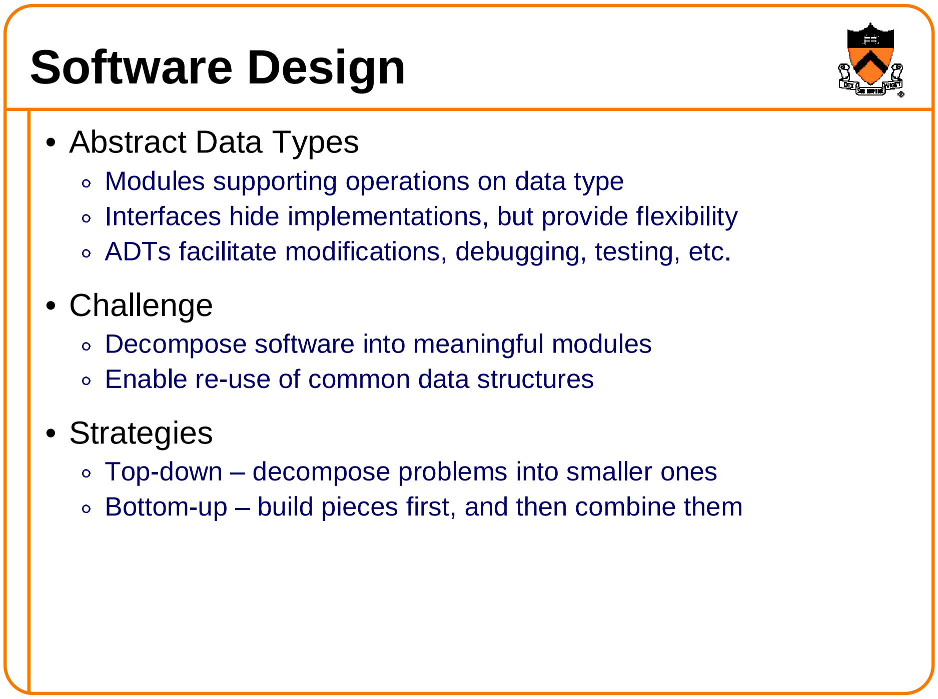


Software Design

(Abstract Data Type Example)

CS 217



Software Design

- Abstract Data Types
  - Modules supporting operations on data type
  - Interfaces hide implementations, but provide flexibility
  - ADTs facilitate modifications, debugging, testing, etc.
- Challenge
  - Decompose software into meaningful modules
  - Enable re-use of common data structures
- Strategies
  - Top-down – decompose problems into smaller ones
  - Bottom-up – build pieces first, and then combine them

StringArray Example

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;
}
```

Dynamic Array ADT

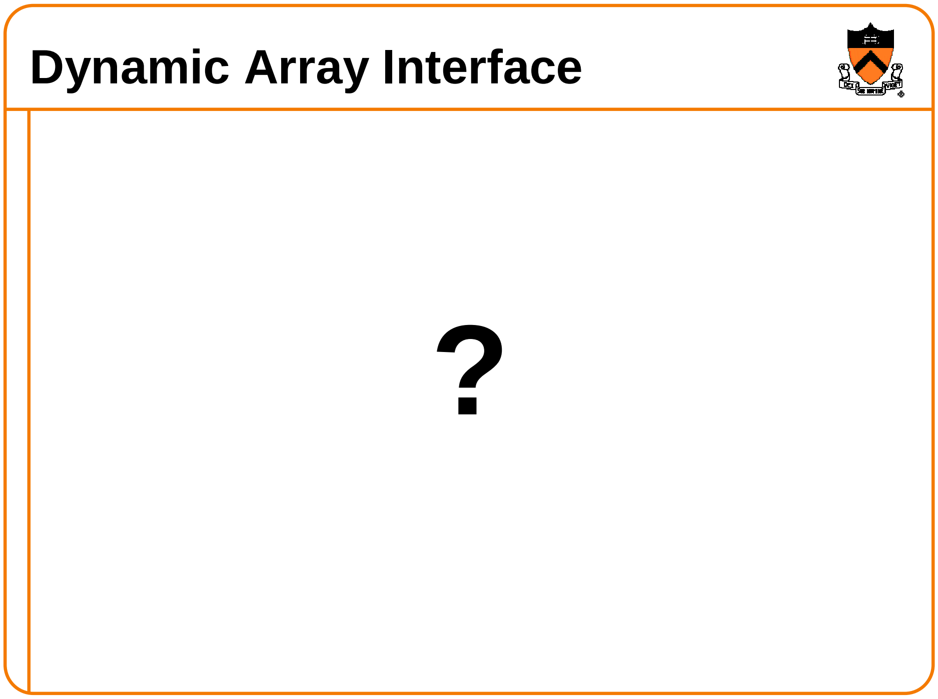
- Goal:
  - Build an array ADT that stores sequences of any type
  - Support add, remove, get, sort, ...
- Advantages over C arrays:
  - Can grow dynamically to any size (with realloc)
  - Can insert in middle (other shifted get moved up)
  - Can provide run-time array bounds checking
- Advantages over ADT arrays of specific types:
  - Can re-use code

Dynamic Array Operations

- Insert elements into array:
  ```
  Array_addLast(array, “Hello”);
  Array_addLast(array, “World”);
  ```
- Remove elements from array:
  ```
  Array_removeFirst(array);
  Array_removeLast(array);
  ```
- Get elements from array:
  ```
  char *s1 = Array_getFirst(array);
  char *s2 = Array_getLast(array);
  char *s3 = Array_getKth(array, k);
  ```
- Manipulate array elements:
  ```
  Array_sort(array, CompareStrings);
  Array_reverse(array);
  …
  ```

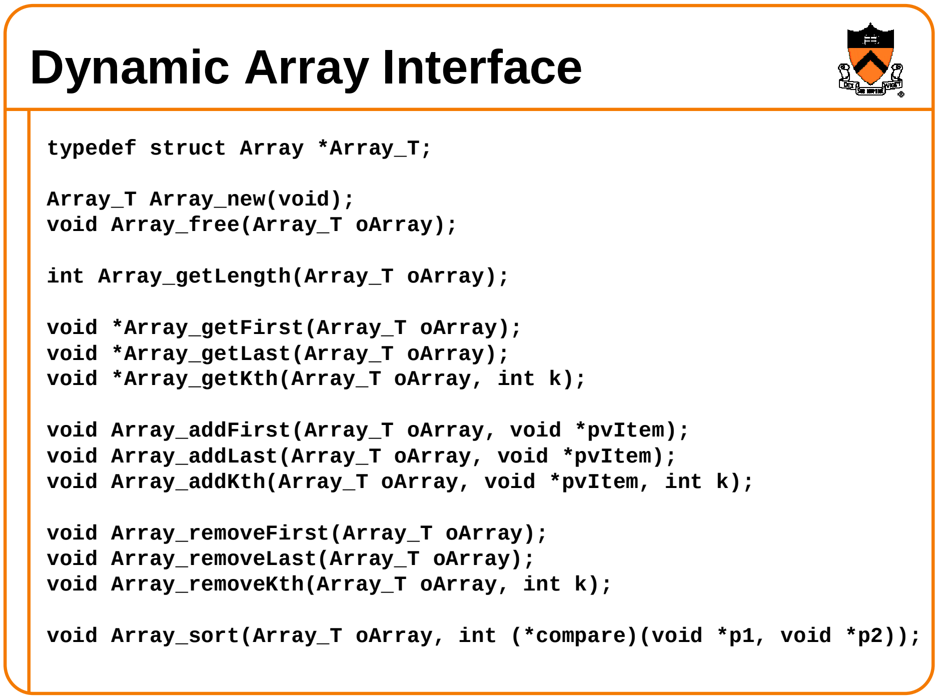
Example: echo

```
int main(int argc, char **argv)
{
  Array_T tokens;
  int i;

  tokens = Array_new();

  for (i = 1; i < argc; i++) {
    Array_add(tokens, argv[i]);
  }

  for (i = 0; i < Array_getLength(tokens); i++) {
    char *token = Array_getKth(tokens, i);
    printf("%s ", token);
  }

  Array_free(tokens);

  return 0;
}
```



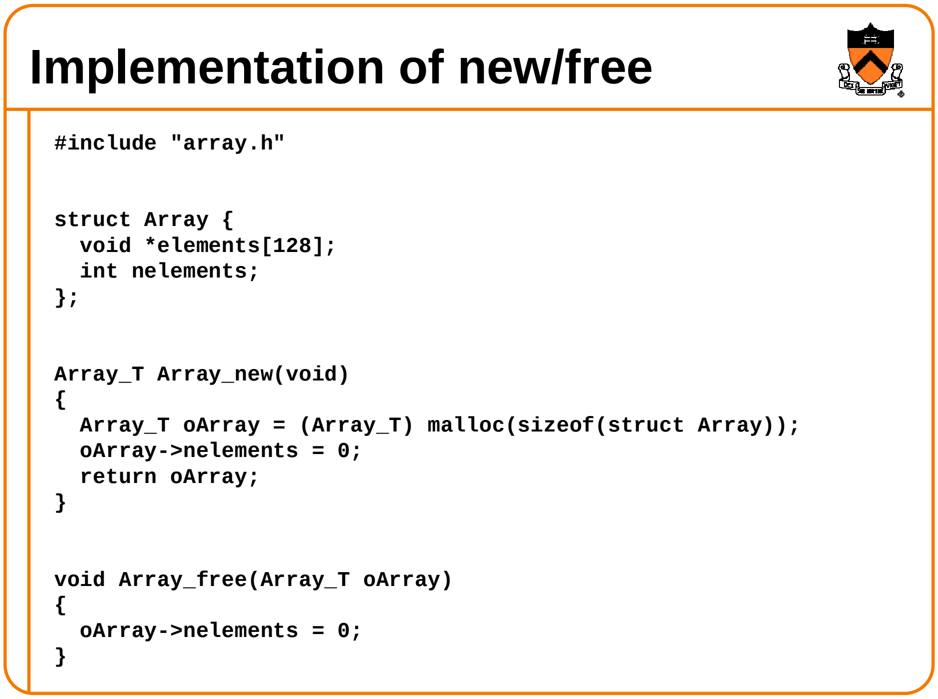
Dynamic Array Interface

?



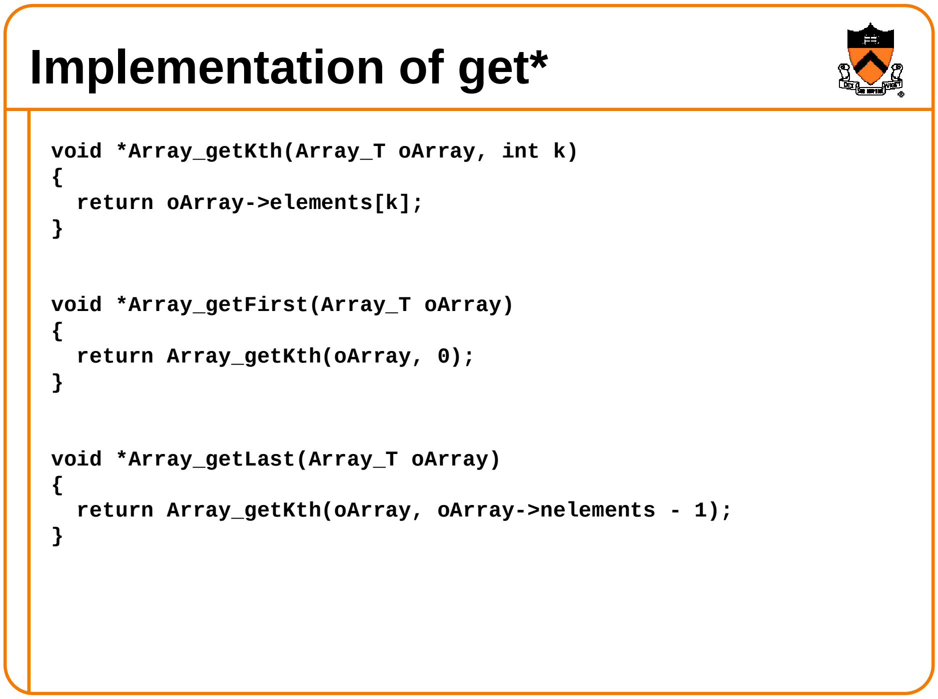
Dynamic Array Interface

```
typedef struct Array *Array_T;

Array_T Array_new(void);
void Array_free(Array_T oArray);

int Array_getLength(Array_T oArray);

void *Array_getFirst(Array_T oArray);
void *Array_getLast(Array_T oArray);
void *Array_getKth(Array_T oArray, int k);

void Array_addFirst(Array_T oArray, void *pvItem);
void Array_addLast(Array_T oArray, void *pvItem);
void Array_addKth(Array_T oArray, void *pvItem, int k);

void Array_removeFirst(Array_T oArray);
void Array_removeLast(Array_T oArray);
void Array_removeKth(Array_T oArray, int k);

void Array_sort(Array_T oArray, int (*compare)(void *p1, void *p2));
```



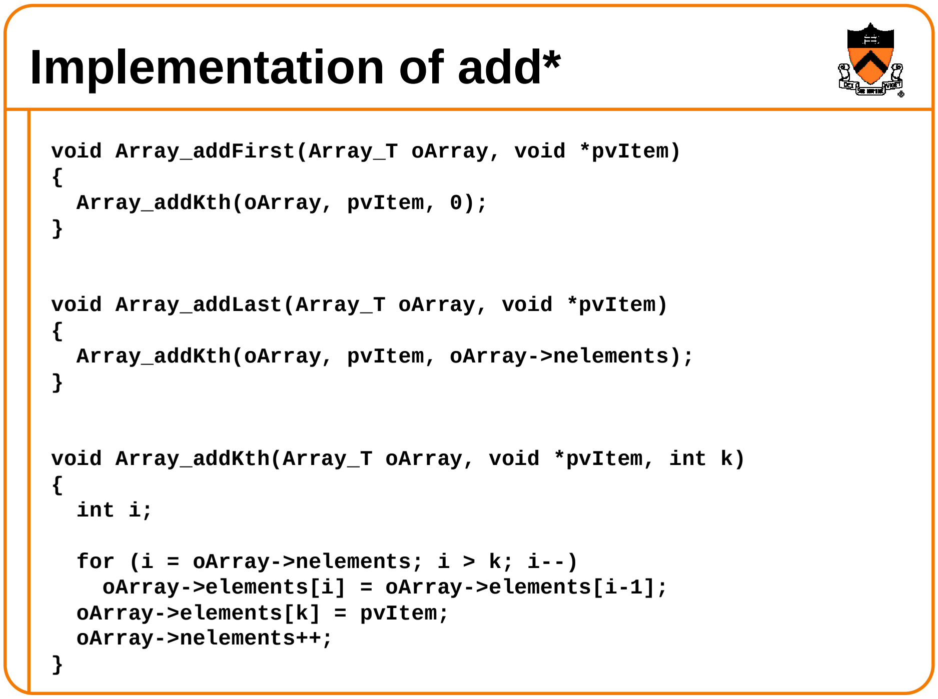
Implementation of new/free

```
#include "array.h"


struct Array {
  void *elements[128];
  int nelements;
};


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


void Array_free(Array_T oArray)
{
  oArray->nelements = 0;
}
```



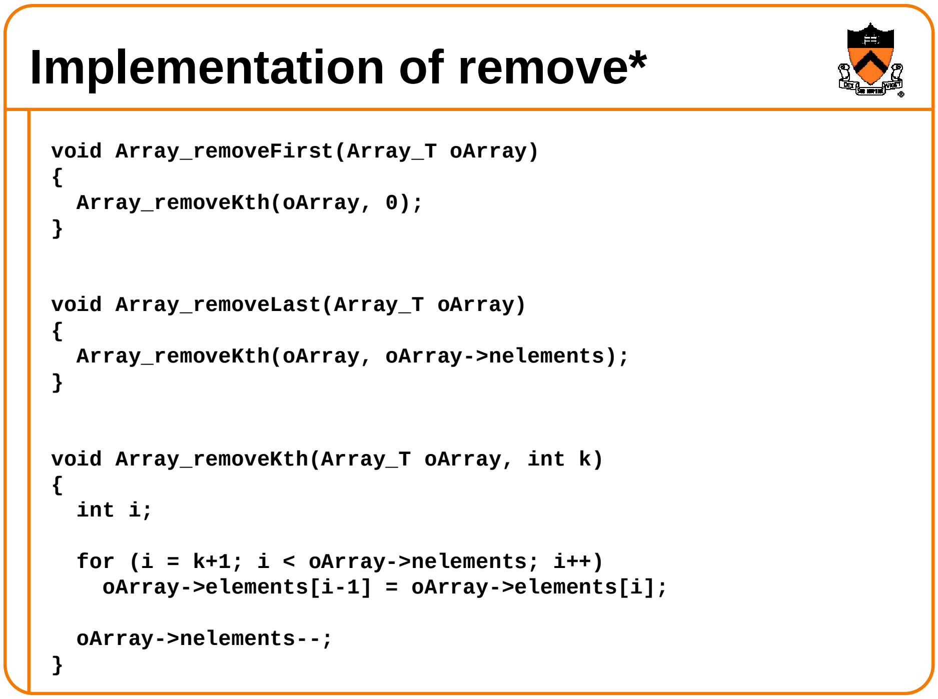
Implementation of get*

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


void *Array_getFirst(Array_T oArray)
{
  return Array_getKth(oArray, 0);
}


void *Array_getLast(Array_T oArray)
{
  return Array_getKth(oArray, oArray->nelements - 1);
}
```



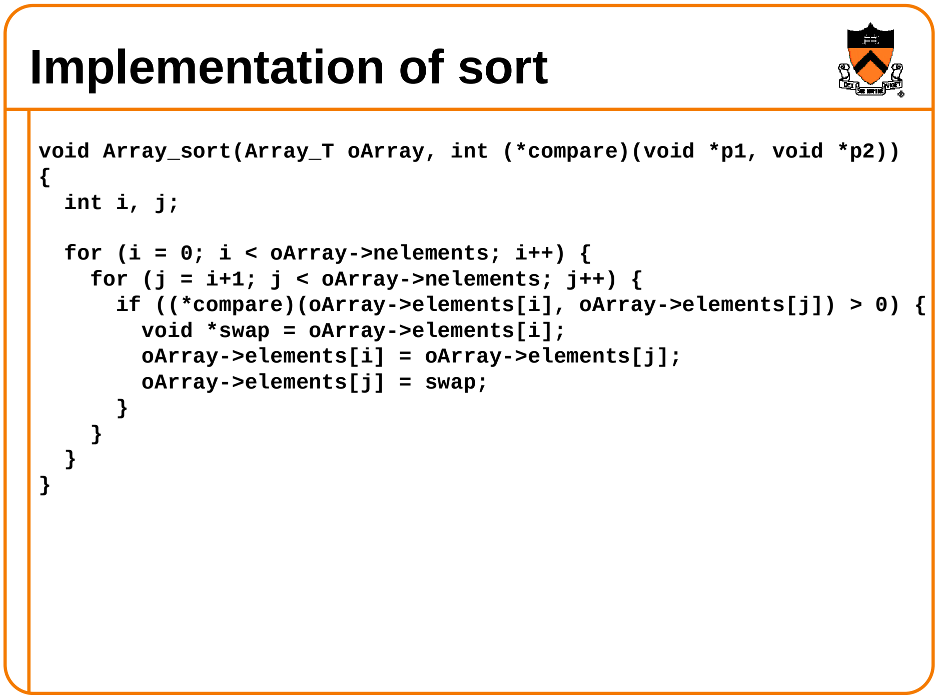
Implementation of add*

```
void Array_addFirst(Array_T oArray, void *pvItem)
{
  Array_addKth(oArray, pvItem, 0);
}


void Array_addLast(Array_T oArray, void *pvItem)
{
  Array_addKth(oArray, pvItem, oArray->nelements);
}


void Array_addKth(Array_T oArray, void *pvItem, int k)
{
  int i;

  for (i = oArray->nelements; i > k; i--)
    oArray->elements[i] = oArray->elements[i-1];
  oArray->elements[k] = pvItem;
  oArray->nelements++;
}
```



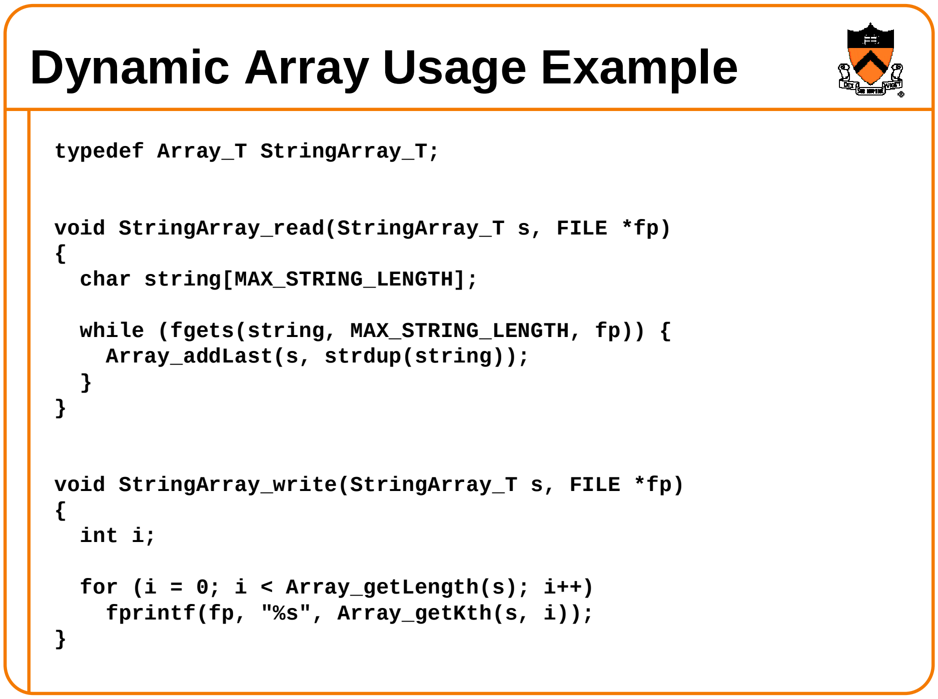
Implementation of remove*

```
void Array_removeFirst(Array_T oArray)
{
  Array_removeKth(oArray, 0);
}


void Array_removeLast(Array_T oArray)
{
  Array_removeKth(oArray, oArray->nelements);
}


void Array_removeKth(Array_T oArray, int k)
{
  int i;

  for (i = k+1; i < oArray->nelements; i++)
    oArray->elements[i-1] = oArray->elements[i];

  oArray->nelements--;
}
```



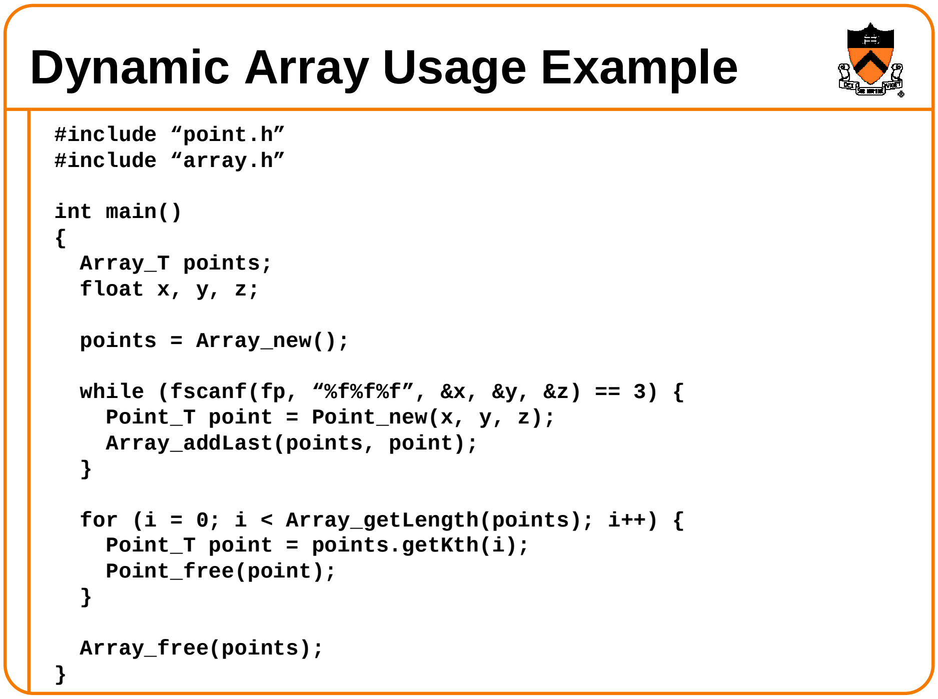
Implementation of sort

```
void Array_sort(Array_T oArray, int (*compare)(void *p1, void *p2))
{
  int i, j;

  for (i = 0; i < oArray->nelements; i++) {
    for (j = i+1; j < oArray->nelements; j++) {
      if ((*compare)(oArray->elements[i], oArray->elements[j]) > 0) {
        void *swap = oArray->elements[i];
        oArray->elements[i] = oArray->elements[j];
        oArray->elements[j] = swap;
      }
    }
  }
}
```



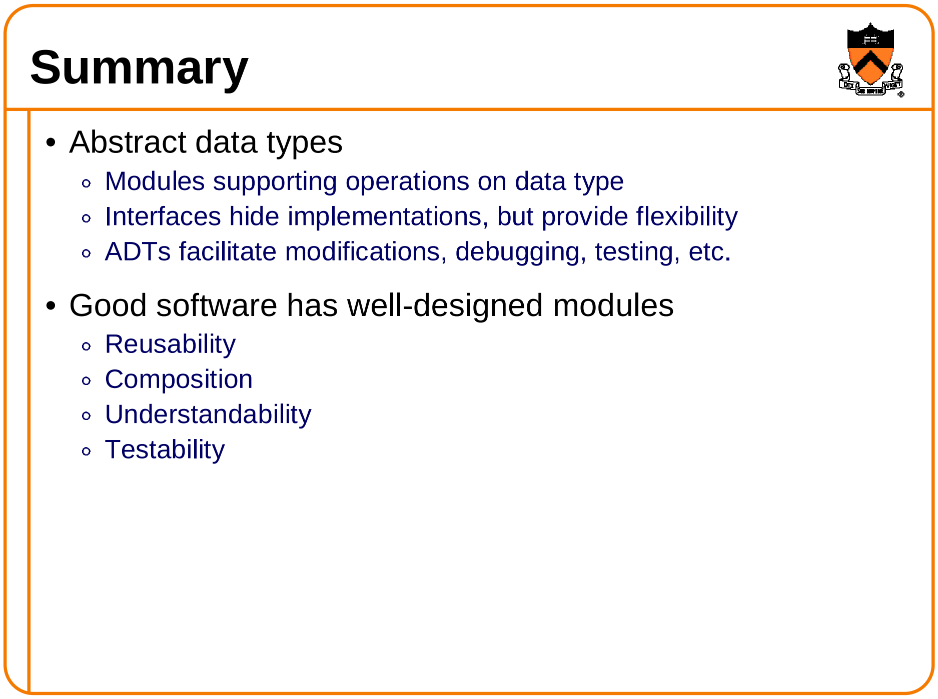
Dynamic Array Usage Example

```
typedef Array_T StringArray_T;


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

  while (fgets(string, MAX_STRING_LENGTH, fp)) {
    Array_addLast(s, strdup(string));
  }
}


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

  for (i = 0; i < Array_getLength(s); i++)
    fprintf(fp, "%s", Array_getKth(s, i));
}
```



Dynamic Array Usage Example

```
#include “point.h”
#include “array.h”

int main()
{
  Array_T points;
  float x, y, z;

  points = Array_new();

  while (fscanf(fp, “%f%f%f”, &x, &y, &z) == 3) {
    Point_T point = Point_new(x, y, z);
    Array_addLast(points, point);
  }

  for (i = 0; i < Array_getLength(points); i++) {
    Point_T point = points.getKth(i);
    Point_free(point);
  }

  Array_free(points);
}
```



Summary

- Abstract data types
  - Modules supporting operations on data type
  - Interfaces hide implementations, but provide flexibility
  - ADTs facilitate modifications, debugging, testing, etc.
- Good software has well-designed modules
  - Reusability
  - Composition
  - Understandability
  - Testability