Module 11: Generic Abstract Data Types and Design

Readings: CP:AMA 19.5, 17.7 (`qsort`)

Generic Data

The void pointer `(void *)` is the closest C has to a "generic" type.

A void pointer is "just" an address; it does not know "what kind of thing" it points at.

We can turn any kind of pointer in a `void *` (even a function), and a `void *` into any kind of pointer.

```c
int add(int a, int b) {
  return a + b;
}
```

```c
int main(void) {
  void *vptr = NULL;
  int i = 136;
  int *pi = &i;
  struct posn my_pos = {3, 4};
  vptr = &add;    // vptr acts as int (*)(int, int)
  vptr = &i;      // vptr acts as int *
  vptr = pi;      // vptr acts as int *
  vptr = &pi;     // vptr acts as int **
  vptr = &my_pos; // vptr acts as struct posn *
  vptr = &vptr;   // vptr acts as void **
  pi = vptr;      // vptr acts as int *
  printf("%d\n", *pi); // prints garbage
}
```

Generic Data

```c
// print_n_add(a, b) Print *a and *b; return sum.
// I sure hope *a and *b are actually int values!
int print_n_add(void *a, void *b) {
  int *ia = a;
  int *ib = b;
  printf("%d + %d = %d\n", *ia, *ib, *ia + *ib);
  return *ia + *ib;
}
```

```c
int main(void) {
  int two = 2;
  int three = 3;
  print_n_add(&two, &three);
}
```

Any time we use a function that takes a `void *`, we need to know what type of value the `void *` "should" point at.

We can't dereference a `void *`, so we need to convert it to a "normal" pointer.

Here `ia` and `ib` are definitely just `int *` values.

Generic Data

We are responsible for ensuring we give things of the right type in `void *` values.

```c
// print_n_add(a, b) Print *a and *b; return sum.
// I sure hope *a and *b are actually int values!
int print_n_add(void *a, void *b) {
  int *ia = a;
  int *ib = b;
  printf("%d + %d = %d\n", *ia, *ib, *ia + *ib);
  return *ia + *ib;
}

int main(void) {
  int two = 2;
  int three = 3;
  print_n_add(&two, &three);
}
```

All sorts of crazy things can happen if we give it pointers to other values.

Exercise Experiment with `print_n_add`: create variables of other types, and pass it pointers to them.

Try strings, structs, all sorts of pointers, ...

Generic algorithms

A “generic” algorithm is an algorithm that can work on values of **any type**.

How can this work? As an example, consider sorting.

To sort, we need to be able to compare two items, and determine which comes first.

We write a function that does this for items in our array, but using `void *` parameters.

```
// Assume v0 and v1 are int;
// determine which comes first.
int int_cmp(const void *v0,
            const void *v1) {
  const int *i0 = v0;
  const int *i1 = v1;
  return *i0 - *i1;
}
```

```
int x = 42;
int y = 6;
trace_int(int_cmp(&x, &y)); ⇒ 36
trace_int(int_cmp(&y, &x)); ⇒ -36
trace_int(int_cmp(&x, &x)); ⇒ 0
```

Now given pointers to two `int` values, we can tell if they are in order.

Generic algorithms

Now that we have the comparison function, we can use `qsort`, from `stdlib.h`.
It will call the comparison function, as needed.

To `qsort`, we pass:

- a pointer to the first item in the array;
- the length of the array;
- the **size of an item** from the array;
- a pointer to a **generic** comparison function.

```
int arr[9] =
  {80, 64, 81, 97, 36, 88, 13, 51, 62};
trace_array_int(arr, 9);
qsort(arr, 9, sizeof(*arr),
      &int_cmp);
trace_array_int(arr, 9);
```

```
>>> [qsort-demo.c|main|52] >> arr => [80, 64, 81, 97, 36, 88, 13, 51, 62]
>>> [qsort-demo.c|main|55] >> arr => [13, 36, 51, 62, 64, 80, 81, 88, 97]
```

Generic algorithms

```c
// Assume v0 and v1 are int;
// determine which comes first.
int int_cmp(const void *v0,
            const void *v1) {
  const int *i0 = v0;
  const int *i1 = v1;
  return *i0 - *i1;
}
```

```c
int arr[9] =
  {80, 64, 81, 97, 36, 88, 13, 51, 62};
trace_array_int(arr, 9);
qsort(arr, 9, sizeof(*arr),
      &int_cmp);
trace_array_int(arr, 9);
```

Exercise

Using `int_cmp` as a model, write a new comparison function so you can sort positive `int` values by their **last digit**.

```c
  qsort(arr, 9, sizeof(*arr), &lastdigit_cmp);
  trace_array_int(arr, 9);
```

```
>>> [qsort-demo.c|main|63] >> arr => [80, 51, 81, 62, 13, 64, 36, 97, 88]
```

Generic algorithms

```
char *words[] = {
  "take", "sulkily", "spenders", "colons", "zwieback", "can", "wit",
  "bug", "ant", "duck","outsell", "judo"
};
int len = 12;
```

Exercise

Write a comparison function so you can sort an array of strings **alphabetically**.

```
qsort(words, len, sizeof(*words), &alphcmp);
```

```
[ ant bug can colons duck judo outsell spenders sulkily take wit zwieback ]
```

Exercise

Write a comparison function so you can sort an array of strings **by length**.
Sort strings of equal length alphabetically.

```
qsort(words, len, sizeof(*words), &lencmp);
```

```
[ ant bug can wit duck judo take colons outsell sulkily spenders zwieback ]
```

Generic algorithms

C also provides a generic **binary search** function `bsearch` that searches any sorted array for a key, and either returns a pointer to the element if found, or `NULL` if not found.

To use it we first create a sorted list:

```
int arr[9] =
  {80, 64, 81, 97, 36, 88, 13, 51, 62};
qsort(arr, 9, sizeof(*arr),
      &int_cmp);
trace_array_int(arr, 9);
```

```
>>> [qsort-demo.c|main|55] >> arr => [13, 36, 51, 62, 64, 80, 81, 88, 97]
```

Then call it, using the same comparison function used for sorting:

```
int target = 51;
assert(bsearch(&target, arr, 9, sizeof(*arr), int_cmp) == &arr[2]); // 51 is in slot 2.
target = 52;
assert(bsearch(&target, arr, 9, sizeof(*arr), int_cmp) == NULL);    // 52 is nowhere.
```

Exercise

write a generic algorithm
`void *generic_max(void *arr, int n, int elsize, int (*f)(void *, void *))` that takes a pointer to an array, the number of items, the size of each item, and a function that compares two.
It returns a `void *` that points to the largest item in the array, as defined by the comparison function.

```
int arr[8] =
  {80, 64, 81, 97, 88, 13, 51, 62};
trace_array_int(arr, 8);
int *maxp = generic_max(arr, 8, sizeof(*arr), &int_cmp);
assert(maxp == &arr[3]);
```

Exercise

Write a generic `map` function that mutates `arr`:

```
void map(void *arr, int arr_len, int elsize, void (*map_func)(void *))
```

Generic Array ADT

We can define an **Array ADT** that the following operations:

1. **create(size)**
2. **destroy()**
3. **set-at(item, index)**
4. **get-at(index)**
5. **print()**

We can implement this in C using the following:

```c
struct array;
struct array *array_create(int length,
    void(*destroy_item)(void *),
    void(*print_item)(const void *));
void array_destroy(struct array *s);
// array_set_at(s, item, index)
// requires: item is a pointer from malloc.
void array_set_at(struct array *s, void *item, int index);
void *array_get_at(const struct array *s, int index);
void array_print(const struct array *s);
```

Here let us plan to have the ADT use `destroy_item` to destroy ever item we give it (using `array_set_at`).

(The **print** operation is provided primarily for debugging, and the `print_item` function pointer is a necessary helper.)

Generic Array ADT

```
struct array *array_create(int length,
          void(*destroy_item)(void *),
          void(*print_item)(const void *)) {
  struct array *s = malloc(sizeof(*s));
  s->data = malloc(length * sizeof(void *));
  for (int i = 0; i < length; ++i)
    s->data[i] = NULL; // start empty
  s->len = length;
  s->print_item = print_item;
  s->destroy_item = destroy_item;
  return s;
}
```

```
void array_set_at(struct array *s,
                  void *item, int index) {
  assert(s && 0 <= index && index < s->len);
  if (s->data[index] != NULL)
    s->destroy_item(s->data[index]);
  s->data[index] = item;
}
```

```
struct array {
  void **data;
  int len;
  void (*print_item)(const void *);
  void (*destroy_item)(void *);
};
```

Exercise

Complete `array_destroy(struct array *s)`, `void *array_get_at(const struct array *s, int index)`, and `void array_print(const struct array *s)`.

Implementing ADTs with void pointers

There are two complications that arise from implementing ADTs with `void` pointers:

- **Memory management** is a problem because a protocol must be established to determine if the client or the ADT is responsible for freeing item data.
- **Comparisons** are a problem because some ADTs must be able to compare items when searching and sorting.

The solution to the **memory management** problem is to make the *ADT interface explicitly clear* whose responsibility it is to `free` any item data: the client or the ADT. Both choices present problems.

For example, when it is the **client's responsibility** to `free` items, care must be taken to retrieve and `free` every item before a `destroy` operation, otherwise `destroy` could cause memory leaks. A precondition to the `destroy` operation could be that the ADT is empty (all items have been removed).

Implementing ADTs with void pointers

When it is the **ADT's responsibility**, problems arise if the items contain additional dynamic memory.

For example, consider if we desire a **sequence of stacks**, where each stack is an instance of the stack ADT. If the sequence `remove_at` operation simply calls `free` on the item, it causes a memory leak as the stack data is not freed.

To solve this problem, the client can provide a customized `free` function for the ADT to call (e.g. `stack_destroy`).

Implementing ADTs with void pointers

example: stack interface with void pointers

```
// (partial interface) CLIENT'S RESPONSIBILITY TO FREE ITEMS

// stack_push(s, i) puts item i on top of the stack
//   NOTE: The caller should not free the item until it is popped
void stack_push(Stack s, void *i);

// stack_top(s) returns the top but does not pop it
//   NOTE: The caller should not free the item until it is popped
const void *stack_top(Stack s);

// stack_pop(s) removes the top item and returns it
//   NOTE: The caller is responsible for freeing the item
void *stack_pop(Stack s);

// stack_destroy(s) destroys the stack
// requires: The stack must be empty (all items popped)
void stack_destroy(Stack s);
```

Implementing ADTs with void pointers

example: client interface

```c
#include "stack.h"

// this program reverses the characters typed
int main(void) {
  Stack s = stack_create();
  while(1) {
    char c;
    if (scanf("%c", &c) != 1) break;
    char *newc = malloc(sizeof(char));
    *newc = c;
    push(s, newc);
  }
  while(!is_empty(s)) {
    char *oldc = pop(s);
    printf("%c", *oldc);
    free(oldc);
  }
  stack_destroy(s);
```

Selecting a data structure

In Computer Science, every data structure is some **combination** of the following “core” data structures.

- primitives (e.g. an `int`)
- structures (i.e. `struct`)
- arrays
- linked lists
- trees
- graphs

Selecting a data structure

Selecting an appropriate data structure is important in **program design**. Consider a situation where you are choosing between an array, a linked list, and a BST. Some design considerations are:

- How frequently will you add items? remove items?
- How frequently will you search for items?
- Do you need to access an item at a specific position?
- Do you need to preserve the “original sequence” of the data, or can it be re-arranged?
- Can you have duplicate items?

Knowing the answers to these questions and the efficiency of each data structure function will help you make design decisions.

Sequenced data

Consider the following strings to be stored in a data structure.

```
"Wei" "Jenny" "Ali"
```

Is the **original order** important?

- If it's the result of a competition, yes: `"Wei"` is in first place.
  We call this type of data *sequenced*.
- If it's a list of friends to invite to a party, it is not important.
  We call this type of data *unsequenced* or "rearrangeable".

If the data is ordered, then a data structure that *sorts* the data (e.g. a BST) is likely not an appropriate choice. Arrays and linked lists are better suited for ordered data.

Data structure comparison: sequenced data

Function	Dynamic Array	Linked List
`item_at`	$O(1)$	$O(n)$
`search`	$O(n)$	$O(n)$
`insert_at`	$O(n)$	$O(n)$
`insert_front`	$O(n)$	$O(1)$
`insert_back`	$O(1)^*$	$O(1)^\dagger$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(1)^\diamond$

* amortized

† requires a `back` pointer: $O(n)$ without

$^\diamond$ requires a *doubly* linked list and a `back` pointer: $O(n)$ without.

Data structure comparison: unsequenced (sorted) data

Function	Sorted Dynamic Array	Sorted Linked List	Regular BST	Self-Balancing BST
`select`	$O(1)$	$O(n)$	$O(h)^{\dagger}$	$O(\log n)^{\dagger}$
`search`	$O(\log n)$	$O(n)$	$O(h)$	$O(\log n)$
`insert`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$
`remove`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$

† requires a `count` augmentation: $O(n)$ without.

`select(k)` finds the item with index k in the structure.
For example, `select(0)` finds the smallest element.

Sequenced data

example: design decisions

- An array is a good choice if you frequently access elements at specific positions (random access).
- A linked list is a good choice for sequenced data if you frequently add and remove elements at the start.
- A self-balancing BST is a good choice for unsequenced data if you frequently search for, add and remove items.
- A sorted array is a good choice if you rarely add/remove elements, but frequently search for elements and select the data in sorted order.

Goals of this Section

At the end of this section, you should be able to:

- determine an appropriate data structure or ADT for a given design problem
- describe the memory management issues related to using `void` pointers in ADTs and how `void` pointer comparison functions can be used with generic ADTs and generic algorithms

An ADT is the combination of data and functions that “operate” on that data.
In *object-oriented programming* (CS 246), a *class* can encapsulate this combination of data and functions (methods).
Classes have additional features (behaviours) that extend beyond ADTs.