Lecture 15:
Abstract Data Types
Logistics

• Visit your test Friday Nov 22, 1015am-1230pm
• There _is_ lab this week
• Next week: weird week
  – Lab canceled
  – No lecture Thurs Nov 28\textsuperscript{th} (Thanksgiving)
• Only 5 lectures left (including this one) ... we have lots to do.
This is the remainder of today’s lecture

• Abstract data types
  – Think about it from the perspective of the user
  – NOT the implementor
Data types

• Simple data types
  – float, double, int, char, unsigned char

• Complex data types
  – Defined with structs

• Abstract data type
  – Accomplished through function calls
  – You don’t have to know the details
Abstract data types

• Two pieces:
  – Define behavior (via function prototypes)
  – Define implementation (via functions)

• You can have more than one implementation for a given behavior
(Bad) Example: One Interface, Multiple Implementations

```c
int TwoStudentsWithSameName(Student *students, int numStudents)
{
    int i, j;
    for (i = 0 ; i < numStudents ; i++)
        for (j = 0 ; j < numStudents ; j++)
        {
            if (i == j)
                continue;
            if (strcmp(students[i].name, students[j].name) == 0)
                return 1;
        }
    return 0;
}
```

Bad example since we are talking about data types ... this is just a function that works on data
Better Example: Store/Fetch

• Abstract data type has two methods:
  – Store
    • Takes a “key” and a “value”
  – Fetch
    • Takes a ”key”, and returns a “value

• Example:
  – Key == UO ID
  – Value == student struct
DETOUR

• We are going to spend the next 15 slides & 30-40 minutes talking about an example.
• This example will then be used to show ADTs.
• The ADT will do store/fetch
• We need two ideas for how to do store/fetch
  – Arrays
  – Hash tables
• Observation:
• I don’t need to know anything about ADT
• How is it implemented?
• How long does Store take?
• How long does Fetch take?

```
#include <string.h>

typedef struct
{
    float score_1A;
    float score_1B;
    /* etc */
} Grades;

typedef struct
{
    char *name;
    Grades grades;
} Student;

{
    Student cis212_F18[80];
    cis212_F18[0].name = "Henry Shields";
    cis212_F18[0].grades.score_1A = 0;
    cis212_F18[0].grades.score_1B = 0;

    AbstractDataType adt;
    Initialize(&adt);
    adt.Store(951000001, cis212_F18+0);
    adt.Store(951000002, cis212_F18+1);
    /* ... */
    Student *s = adt.Fetch(951000001);
    /* s is now Henry Shields */
}
To Motivate ADT, We Need Examples of Data Structures That Can Do Store/Fetch

• Two Examples:
  – Array
  – HashTable
One Data Structure for Store/Fetch: Array

- Observation:
- Not very generic (int key, Student value)

Why not pass in StoreFetchArray * instead of void *?
A: need this later

```c
#define MAX_STUDENTS 1000

typedef struct
{
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
    int curID;
} StoreFetchArray;

void ArrayInitialize(StoreFetchArray *arr)
{
    arr->curID = 0;
}

void ArrayStore(void *a, int key, Student *v)
{
    StoreFetchArray *arr = (StoreFetchArray *) a;
    if (arr->curID >= MAX_STUDENTS)
        exit(EXIT_FAILURE);

    arr->keys[arr->curID] = key;
    arr->values[arr->curID] = v;
    arr->curID++;
}

Student *ArrayFetch(void *a, int key)
{
    StoreFetchArray *arr = (StoreFetchArray *) a;
    for (int i = 0; i < arr->curID; i++)
        if (arr->keys[i] == key)
            return arr->values[i];
    return NULL;
}
Complexity for Array

- Store: $O(1)$
- Fetch: $O(N)$
HashTable

• Idea:
  – Create a big array with keys and values
    • (Just like last slide!)
  – But: don’t insert starting from the beginning
  – Instead: insert into “random” places in the array
  – Not truly random, as it needs to be reproducible
  – Typical: take key and perform some math operation on it

This will be a whirlwind intro to hash tables. We will return to this idea later.
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>
HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
UO_ID: 951001234
Student: “xFF”

Idea:
Turn UO_ID into an index.
In this case, %10.
(much more complex ideas)
## HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>..34</td>
<td>-1</td>
<td>..66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

**Store:**
- UO_ID: 951003266
- Student: “xAF”
### HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>..66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

#### Store:
- **UO_ID:** 951012344
- **Student:** “xAA”

#### Idea:
- Slot 4 is full ... just use the next slot (slot 5).
- Keep going until you find one, including wraparounds
## HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
UO_ID: 951012345
Student: “xB8”
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

**Fetch:**

UO_ID: 951045323
EASY: NULL
HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951012345
More work ... walk from index 5 to 6 to 7 ... found it!
**HashTable**

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951012355
More work ... walk from index 5 to 6 to 7 to 8... not there!
Another Data Structure for Store/Fetch: Hash Table

```c
typedef struct
{
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
} HashTable;

void HashTableInitialize(HashTable *arr)
{
    for (int i = 0; i < MAX_STUDENTS; i++)
    {
        arr->keys[i] = -1;
        arr->values[i] = NULL;
    }
}

void HashTableStore(void *a, int key, Student *v)
{
    HashTable *ht = (HashTable *) a;
    for (int i = 0; i < MAX_STUDENTS; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == -1)
        {
            ht->keys[idx] = key;
            ht->values[idx] = v;
            return;
        }
    }
    exit(EXIT_FAILURE);
}

Student *HashTableFetch(void *a, int key)
{
    HashTable *ht = (HashTable *) a;
    for (int i = 0; i < MAX_STUDENTS; i++)
    {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == key)
            return ht->values[idx];
        if (ht->keys[idx] == -1)
            return NULL;
    }
    return NULL;
}
```

• Observation:
• Still not very generic.
  (int key, Student value)
Complexity for Hash Table

• Store: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(n)$

• Fetch: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(N)$

• Gets into new topic ... expected performance.
To Motivate ADT, We Need Examples of Data Structures That Can Do Store/Fetch

• Two Examples:
  – Array
  – HashTable
Dispatch Table

- Dispatch table: allows for “methods” to be define on ADT
- Done with a struct that contains pointers to functions

```c
/* now define the dispatch table */
struct theClass {
    /* the private data of the instance */
    void *self;

    /* function pointer to method a() */
    void (*a)(const TheClass *tc);

    /* function pointer to method b() */
    void (*b)(const TheClass *tc);
};
```
typedef struct 
{
    void    *self;
    void     (*Store)(void *, int, Student *);
    Student * (*Fetch)(void *, int);
} StoreFetchADT;

void ADTStore(StoreFetchADT *adt, int key, Student *s)
{
    adt->Store(adt->self, key, s);
}

Student *ADTFetch(StoreFetchADT *adt, int key)
{
    return adt->Fetch(adt->self, key);
}
Constructing ADT from StoreFetchArray or HashTable

```c
StoreFetchADT *CreateArrayStoreFetchADT()
{
    StoreFetchArray *sfa = malloc(sizeof(StoreFetchArray));
    ArrayInitialize(sfa);
    StoreFetchADT *adt = malloc(sizeof(StoreFetchADT));
    adt->self = sfa;
    adt->Store = ArrayStore;
    adt->Fetch = ArrayFetch;
    return adt;
}

StoreFetchADT *CreateHashTableStoreFetchADT()
{
    HashTable *ht = malloc(sizeof(HashTable));
    HashTableInitialize(ht);
    StoreFetchADT *adt = malloc(sizeof(StoreFetchADT));
    adt->self = ht;
    adt->Store = HashTableStore;
    adt->Fetch = HashTableFetch;
    return adt;
}
```

This is why ArrayStore/ArrayFetch needed to be void *
int main()
{
    StoreFetchADT *sf_adt = CreateArrayStoreFetchADT();
    Student s1;
    s1.name = "Hank Childs";
    Student s2;
    s2.name = "Henry Childs";
    ADTStore(sf_adt, 37, &s1);
    ADTStore(sf_adt, 42, &s2);
    Student *s = ADTFetch(sf_adt, 37);
    printf("Student is %s\n", s->name);
    s = ADTFetch(sf_adt, 3);
    printf("Student is %p\n", s);
    s = ADTFetch(sf_adt, 42);
    printf("Student is %s\n", s->name);
}

Hanks-iMac:212 hank$ gcc adt.c
Hanks-iMac:212 hank$ ./a.out
Student is Hank Childs
Student is 0x0
Student is Henry Childs
Data structure for specific data type (example: array of Students)

Data structure for *any* data type (example: array of “void *”)

Abstract data structure for specific data type (example: search for student names)

Abstract data structure for *any* data type (example: search for “void *”)
When Do I Use What?

- When I have developed a data structure I want others to use

  - Data structure for specific data type (example: array of Students)

  - Data structure for *any* data type (example: array of “void *”)

- When I want to solve a problem for many, many people

  - Abstract data structure for specific data type (example: search for student names)

  - Abstract data structure for *any* data type (example: search for “void *”)

- When writing something for my specific program

  - Never? (but also what we just did)
Specific data structure for specific type

Specific data structure for generic type

Abstract data types

No management/organization of data
File adt.c is online. Let’s look at it for a while.
Project 3A

• 3A is a little mismatched.
• I am referring to “Fruit” as a dispatch table, but note that Fruit is not really an abstract data type. It is more an “abstract type.”
• But the machinery (dispatch table) is the same.
Project 3A

CIS 212: Project #3A
Assigned: November 19, 2019
Due: November 26, 2019
(which means submitted by 6am on November 27, 2019)
Worth 9% of your grade
(note: there will be another assignment on November 21, so you will be working on
two assignments at once)

You will be implementing a dispatch table. The dispatch table will be called “Fruit”. It will allow us to deal with fruits abstractly. You will also implement Banana and Apple structs.

The Fruit struct will contain function pointers. One function pointer will point to a function that returns the volume of a fruit. The other function pointer will point to a function that returns the color of a fruit.
Project 3A

Fruit *CreateApple(double r, int isGreen)
{
    Apple *a = malloc(sizeof(Apple));
a->radius = r;
a->green = isGreen;
Fruit *f = malloc(sizeof(Fruit));
f->self = a;
f->GetColor = AppleGetColor;
f->GetVolume = AppleGetVolume;
return f;
}
Stacks
Stacks

• A data structure
• 2 methods: push and pop
• Sometimes a third: peek
Example: Stack of Integers

```c
#define MAX_STACK_SIZE 100

typedef struct
{
    /* your data members go here */
} Stack;

void Initialize(Stack *s)
{
}

void Push(Stack *s, int X)
{
}

int Pop(Stack *s)
{
}

int main()
{
    Stack s;
    int X;
    Initialize(&s);
    Push(&s, 5);
    Push(&s, 6);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    Push(&s, 7);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
}
```

Hanks-iMac:3B hank$ ./a.out
Stacked popped 6
Stacked popped 7
Stacked popped 5
Stack: Asymptotic Complexity

- Push: O(1)
- Pop: O(1)
- Store: O(1)
- Fetch: O(n)

→ pop each element and look, and then restore?