Abstract Data Types

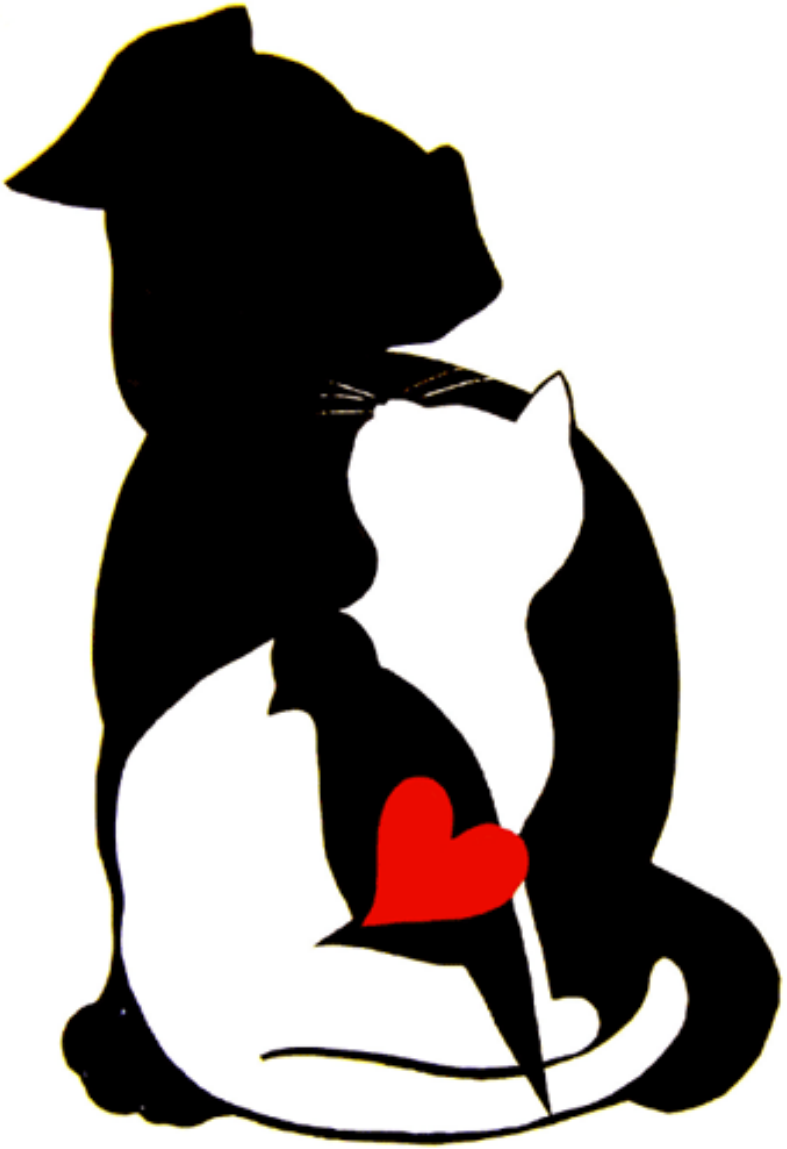
Devin J. Pohly <djpohly@cse.psu.edu>

Abstract data types

- ADTs are essentially encapsulated data structures
  - Most basic examples: list, stack, priority queue, etc.
  - User doesn't need to know how they are implemented
  - Provide a convenient interface
- But the same concepts can be extended to suit many of the most common *object-oriented* tasks!
  - Instance variables and methods
  - "Class" variables and methods
  - Inheritance
  - Even polymorphism!

The classic example

- The animal class
  - Instance variables for name and age
  - Instance method speak
  - Subclasses dog, cat, and bird
    - Each overrides speak with its own method



Instance variables

- Use a *struct* to define any data that belongs to individual instances
  - Side note: this is why `struct` and `class` are virtually synonymous in C++
- Will use pointers to refer to instances
  - The arrow operator will be very useful!

```
struct animal {
     char *name;
     int age;
};

struct cat {
     char *name;
     int age;
     int is_declawed;
};

struct dog {
     char *name;
     int age;
     float bark_decibels;
};

struct bird {
     char *name;
     int age;
     enum marking_type markings;
};
```

Instance methods

- For instance methods, create a function that takes a *pointer to the struct*
  - In many languages, this pointer is called `this`
- You can then use the pointer just as you would use `this` in C++
- To avoid naming collisions, use a common prefix for the function names

```c
struct animal {
     char *name;
     int age;
};

void animal_print_name(struct animal *this)
{
     printf("My name is %s\n", this->name);
}

int animal_set_age(struct animal *this,
          int age)
{
     if (age < 0)
          return 1;
     this->age = age;
     return 0;
}
```

What about constructors?

- Constructors and destructors are really just instance methods
  - Typically called `init` and `destroy`
  - In C, the caller will do the allocation/deallocation of the object
  - These methods should **not** allocate or free the `this` pointer!
  - But they should allocate and free any data needed within the struct

```c
struct animal {
     char *name;
     int age;
};

int animal_init(struct animal *this,
          char *name, int age)
{
     char *newname = strdup(name);
     if (newname == NULL)
          return 1;

     this->name = newname;
     this->age = age;
     return 0;
}

void animal_destroy(struct animal *this)
{
     free(this->name);
}
```

Flexible allocation

- The caller can allocate the object on the stack, heap, or data segment
  - More flexible functionality
  - Remember to *destroy the object* before it goes out of scope!

```
struct animal a1;

void test_allocs(void)
{
     // Already allocated in data segment
     // (note: error handling omitted)
     animal_init(&a1, "Bo", 6);

     // Allocate on the stack
     struct animal a2;
     animal_init(&a2, "Fifi", 3);

     // Allocate on the heap
     struct animal *a3 = malloc(
               sizeof(struct animal));
     animal_init(a3, "Fido", 1);

     // Then, for example:
     animal_print_name(&a1);
     animal_print_name(&a2);
     animal_print_name(a3);

     animal_destroy(a3);
     free(a3);
     animal_destroy(&a2);
     animal_destroy(&a1);
}
```

Class variables/methods

- Remember: class variables and methods are not tied to any one instance
  - No `this` pointer for functions
- These can just be implemented as global variables and functions
  - Not inside the instance struct
  - Statically allocated (hence the use of the `static` keyword for these in other languages)

```
int animal_count = 0;

void animal_add_count(int n)
{
     animal_count += n;
}
```

Information hiding

- It is often a good idea to hide the implementation details
  - Similar to private/protected in OO
  - Keeps user from accidentally messing with the inner workings
- You can do this in C with an *incomplete type* (or *opaque type*)
  - Type for which C doesn't know the size or contents
  - Pointers to incomplete types are OK (C knows how big a pointer is)
  - Declare a struct without an implementation in the public header file and keep the implementation internal
- Then use accessor functions to regulate access to protected data

```
// In animal.h (public header):

struct animal;

int animal_set_age(struct animal *this,
          int age);
```

```
// In animal.c (internal implementation):

struct animal {
     char *name;
     int age;
};

int animal_set_age(struct animal *this,
          int age)
{
     // ...
}
```

Inheritance

- Back to our subclasses...
  - Note the repetition of fields from the superclass
  - Can we do something about this?

```
struct animal {
     char *name;
     int age;
};

struct cat {
     char *name;
     int age;
     int is_declawed;
};

struct dog {
     char *name;
     int age;
     float bark_decibels;
};

struct bird {
     char *name;
     int age;
     enum marking_type markings;
};
```

Inheritance

- Back to our subclasses...
  - Note the repetition of fields from the superclass
  - Can we do something about this?
- Include the superclass as the first member!
  - Remember: structs can contain any complete type, including other structs

```
struct animal {
     char *name;
     int age;
};

struct cat {
     struct animal super;
     int is_declawed;
};

struct dog {
     struct animal super;
     float bark_decibels;
};

struct bird {
     struct animal super;
     enum marking_type markings;
};
```

Working up the chain

- Now we can get fields of the superclass by using the `super` member
  - This is a whole struct, not a pointer, so use a `.` to get its members, not `->`
  - Implementing functions for a subclass is the simple case
  - Note: this is simple single inheritance; multiple inheritance is far more complex

```
struct animal {
     char *name;
     int age;
};

struct cat {
     struct animal super;
     int is_declawed;
};

void cat_print_declawed(struct cat *this)
{
     if (this->is_declawed)
          printf("%s is declawed\n",
                    this->super.name);
     else
          printf("%s is not declawed\n",
                    this->super.name);
}
```

Superclass functions

- Implementing functions for the superclass is harder
  - Have to be able to take instance of any subclass as the `this` parameter

```
struct animal {
     char *name;
     int age;
};

struct cat {
     struct animal super;
     int is_declawed;
};

int animal_can_vote(struct animal *this)
{
     return (this->age >= 18);
}

void can_fluffy_vote(void)
{
     struct cat f;
     cat_init(&f, "Fluffy", 20, 1);

     if (/* what goes here?? */)
          printf("Fluffy can vote\n");

     cat_destroy(&f);
}
```

Superclass functions

- Implementing functions for the superclass is harder
  - Have to be able to take instance of any subclass as the `this` parameter
  - One option: address of the superclass struct

```
struct animal {
     char *name;
     int age;
};

struct cat {
     struct animal super;
     int is_declawed;
};

int animal_can_vote(struct animal *this)
{
     return (this->age >= 18);
}

void can_fluffy_vote(void)
{
     struct cat f;
     cat_init(&f, "Fluffy", 20, 1);

     if (animal_can_vote(&f.super))
          printf("Fluffy can vote\n");

     cat_destroy(&f);
}
```

Superclass functions

- Implementing functions for the superclass is harder
  - Have to be able to take instance of any subclass as the `this` parameter
  - One option: address of the superclass struct
  - Another option: cast the pointer!

```
struct animal {
     char *name;
     int age;
};

struct cat {
     struct animal super;
     int is_declawed;
};

int animal_can_vote(struct animal *this)
{
     return (this->age >= 18);
}

void can_fluffy_vote(void)
{
     struct cat f;
     cat_init(&f, "Fluffy", 20, 1);

     if (animal_can_vote(
               (struct animal *) &f))
          printf("Fluffy can vote\n");

     cat_destroy(&f);
}
```

Casting pointers

- Changes the pointer type but leaves the address alone
  - Warning: **breaks type safety,** so only use if you know what you're doing!
  - Used in the standard library (e.g., for networking protocols)
- Why can we safely cast to the superclass?

Function polymorphism

- Try to figure out how you would implement

```
void animal_speak(struct animal *a)
```

  - Each animal should be able to implement its own speak function (e.g. `cat_speak`), and `animal_speak` should automatically call the appropriate one
  - You should be able to create new animals *without changing `animal_speak`!*
  - Hint: you will need to use function pointers to accomplish this...

- Note: this is very subtle and tricky – consider it an enrichment exercise for the reader!