Lecture Notes,

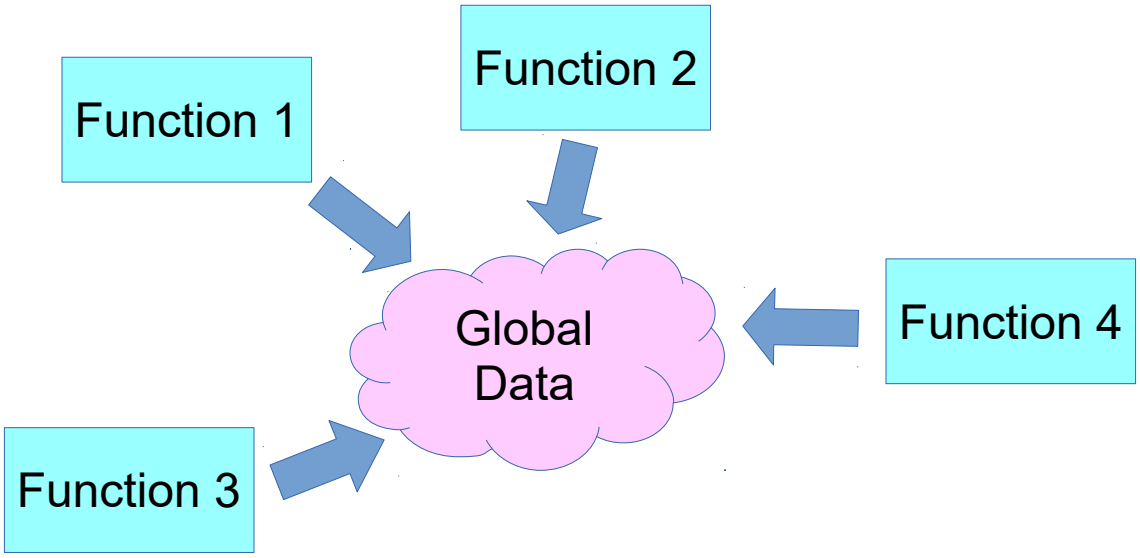
CS345, Winter 2018

Introduction to Object-Oriented Programming

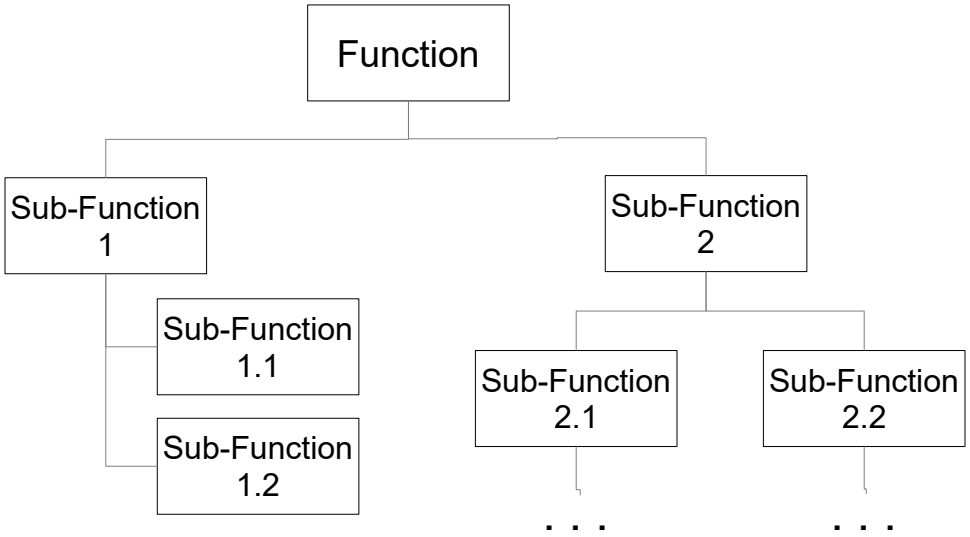
Part I: History and Motivation

Structured Programming

"Structured Programming" (1970s) codified *Procedural Programming*. In procedural programming data—other than local data—is external to the program. Here's a picture:

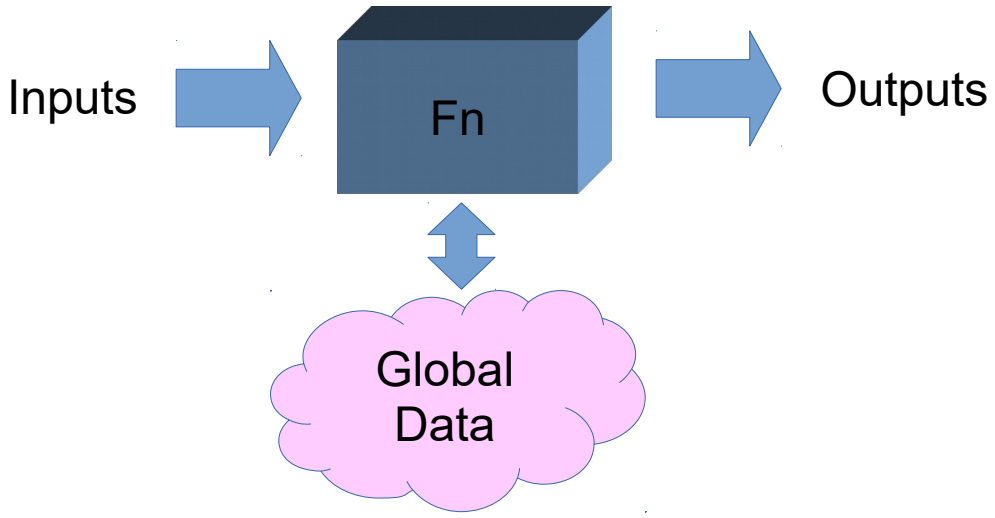


Design of a system was usually done by *functional decomposition*. Functions were broken down into sub-functions, sub-sub-functions, etc. Here's a picture of the idea:



Note that the picture above doesn't tell you how the components interact, it just organizes them hierarchically.

Individual functions are consider to be *black boxes*.



The important idea is that (1) a function transforms inputs to outputs and (2) we don't need to know the internals of how the function does that. This gives us a certain kind of *abstraction*. We don't care how it works inside, only that it correctly transforms its inputs to the correct outputs. Whatever algorithm is used to do the transformation is hidden inside the box. Ideally, other functions cannot make use of knowledge of how the transformation is accomplished.

The function can read and modify the global data. That can cause the function to work differently—produce different outputs for the same inputs—if the global data changes between invocations of the function.

Functional decomposition is a good strategy for analyzing complex functions. Unfortunately, the "structured programming" strategy creates some big problems:

- A bug can create inconsistencies in the global data. If dates are stored as three `int`s, month, day and year, an error might result in a date of February 31, 2018.
- Even in "correct" code, there is no easy way to tell what portion of the global data is important to a specific function and what functions might modify it.

These problems result from the fact that you have no control over which functions access and/or modify the global data. This adds to the difficulty of debugging and correctly modifying the software. These problems reflect problems with the understandability and modifiability of the code.

Another way to say the same thing: All the functions are *coupled* through the shared data. For example, if there is a date stored in the global data, all functions have to agree on how a date is stored: three integers (month, day, year), or some other way.

Why Objects

An Example

Objects grew from as a solution to a common problem. Consider a game like SimCity. The game is played on a grid. There are many kinds of things that can occupy a grid location:

- Dirt—open land
- Residential Zone—people live here
- Power Plant—provides Power to other kinds of things that need power

There are certain common characteristics: For example, where the thing is located in the grid, how many people live there, and how it is displayed as part of the game.

A design/programming problem: Write a function that produces a String describing a location. This might be used, for example, to provide a message about the location when the mouse hovers over the location.

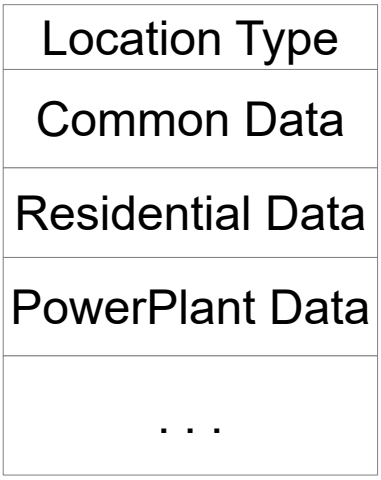
Design/strategy problem: What approach do you use to handle the commonalities and differences between the different kinds of locations in a game?

How do you represent the Data?

Here are some options.

Option 1: One big Record

This is the simplest (and most obvious?) way to handle the problem. Here's a picture and typical C code:




```
struct LocationData {
    int location_type;

    /* Common Data Attributes*/
    . . .
    /* Residential Attributes */
    . . .
    /* PowerPlant Attributes */
    . . .
};
```

Only one of Residential Data, PowerPlant Data, etc. is valid depending on the Location Type.

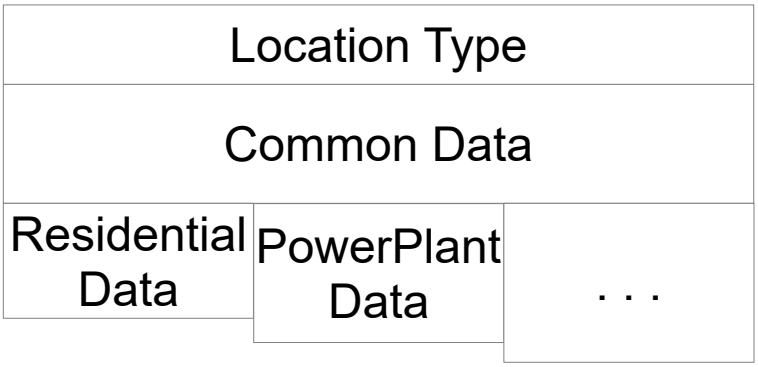
Question: What values, for example, do PowerPlant attributes have when the type is Residential or something else?

Problem: There's nothing to stop you (your program) from referencing Residential Data attributes from code that is expecting other types of locations.

This approach should be considered as a thought experiment. It's unlikely that you would actually do it this way.

Option 2: A Variant Record

Picture and code:




```
struct LocationData {
    int location_type;

    /* Common Data Attributes*/
    . . .

    union {
        struct ResidentialData {
            /* Residential Attributes */
        } res;
        struct PowerPlantData {
            /* PowerPlant Attributes */
        } power;
        . . .
    } type_data;
};
```

Notes (C language): The C `union` statement overlays the data associated with each of its declared members. In this case, each of the members is a `struct` which is why we get a layout like the picture.

Only one of Residential Data, PowerPlant Data or whatever is present depending on Location Type. Depending on the programming language, programs may (Ada) or may not (C and C++) be blocked from accessing data that is inappropriate for the type of

location, for example, accessing power plant data when the location is residential. Note that in C `sizeof(struct LocationData)` would be the largest of the sizes of each of alternatives in the `union`, which may or may not be what you want.

In Ada, this is a variant record. In Python, this would be implemented by only defining the attributes appropriate to the location type.

This fixes some of the problems with Option 1.

The Code

You want to implement a function that returns a String with the description of a location. For both option 1 and option 2, above, your code—in C—will look something like:

```
string describe_location(struct LocationData *loc) {
    switch(loc->location_type) {
        case RESIDENTIAL:
            return describe_residential(loc);
        case POWER_PLANT:
            return describe_powerplant(loc);
        . . .
    }
}
```

Where `describe_residential`, `describe_powerplant`, etc. are functions that provide the correct description for that type of location.

Notes (C language): (1) The parameter to the `describe_location` function is a pointer to a `LocationData` structure, not a copy of the structure. That means that, like Java and Python, modifications to any part of the structure will be visible to the caller. (2) We are assuming that `RESIDENTIAL`, `POWER_PLANT`, etc. are constants that have been defined elsewhere. (3) The `switch` doesn't need break statements since the return will end the function at that point.

Note that in C there is nothing that stops you from mistakenly passing a `RESIDENTIAL` structure to `describe_powerplant`, which would, at best, produce "exciting" output.

Behavior

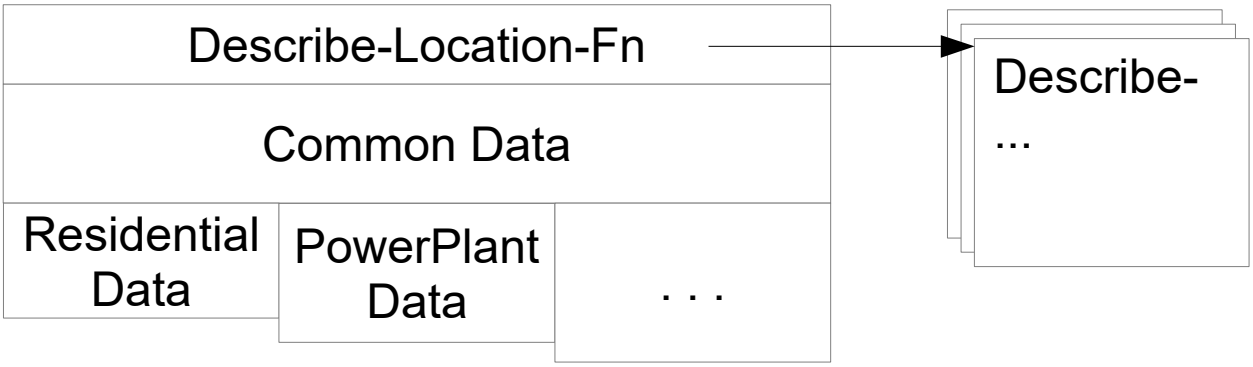
The key idea is that the different kinds of locations all share common *behavior*—having a description—with different details depending on the kind of location.

What is a behavior?

Answer: A set of code. That is, a set of functions with the same names and parameters (signatures) but differing implementations depending on the actual type of the location. For example, the sentence "Part of every location's behavior is a String describing the location" is the same as saying "For each kind of location there is a function that given a specific location of that kind returns a String with the description for that specific location."

It must be noted that the concept of behavior includes not just the signature of the function but also includes additional information about the intent or semantics of the function. In this case, for example, the string is supposed to be a description, which probably includes the idea that the string in not null or the empty string.

Another Approach




```
struct LocationData;

typedef string (*describe_loc_fn) (struct LocationData *);

struct LocationData {
    describe_loc_fn describe_location;

    /* remainder as before */
};
```

Notes (C language): (1) You have to introduce the fact that `LocationData` is a `struct` before you can use that in the following declaration of `describe_loc_fn`. (2) The `typedef` declares `describe_loc_fn` as a pointer to a function that accepts a single parameter pointer to a `LocationData` `struct` and returns a `string`. (3) The `struct LocationData` starts with a pointer to the appropriate describe function.

In this approach, the Location Type is replaced by a reference to a Describe-... function, one each for each kind of location. The code for Describe-Location now looks like:

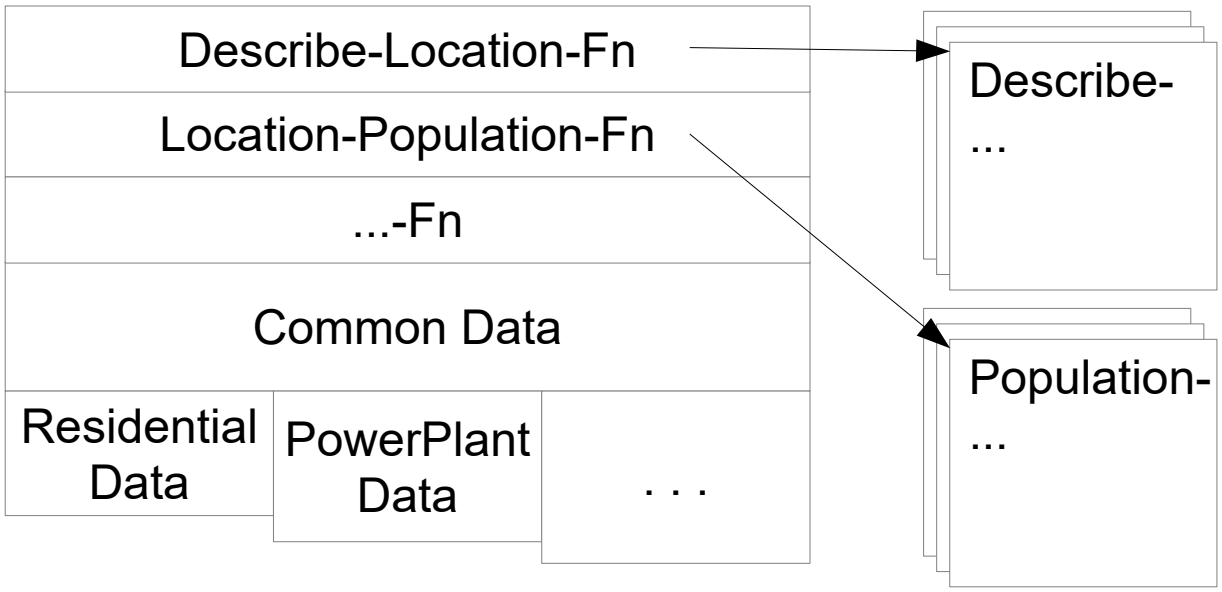
```
loc->describe_location(loc)
```

This is obviously a big improvement. Over and above the reduction in the size of the code (many lines to one), this approach also improves the modifiability of the program. In particular, what if you want to add a new type of location?

- You still have to implement the new Describe-... function. (There is no free lunch.)
- But, the client code doesn't have to change. Compare this to the code for Options 1 and 2 where you have to add a new case and call the correct Describe-... function.

A Problem

This structure still has a problem. There is never a single behavior. A location in the game will have many behaviors. For example, in addition to describing a location, I can ask if the location has power, ask for the population of a location, etc.. This will result in a structure that looks like this:

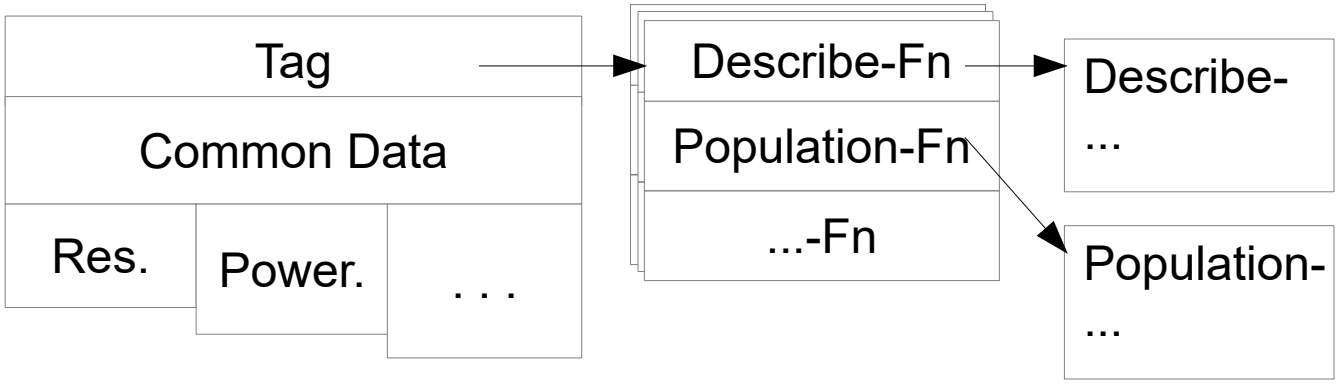


In this structure there are lots of pointers that are all pointing to the same sets of functions. Further, we've introduced another source of possible error: What happens if the pointers get mixed up in some strange way, for example, describing a power plant but getting population for residential. Then if the user hovers the over the location they will get a description of a power plant, However, when counting population, the will likely get inconsistent results, at best, or a spectacular program failure, at worst.

Solution

What do you do when you have a bunch of data that is repeated frequently but with a (relatively) small number of possible sets of values or combinations?

Answer: Make a new structure which holds the data and refer to it (point to it) when you need to. This is a variant of a design pattern called Flyweight. Here's the picture:

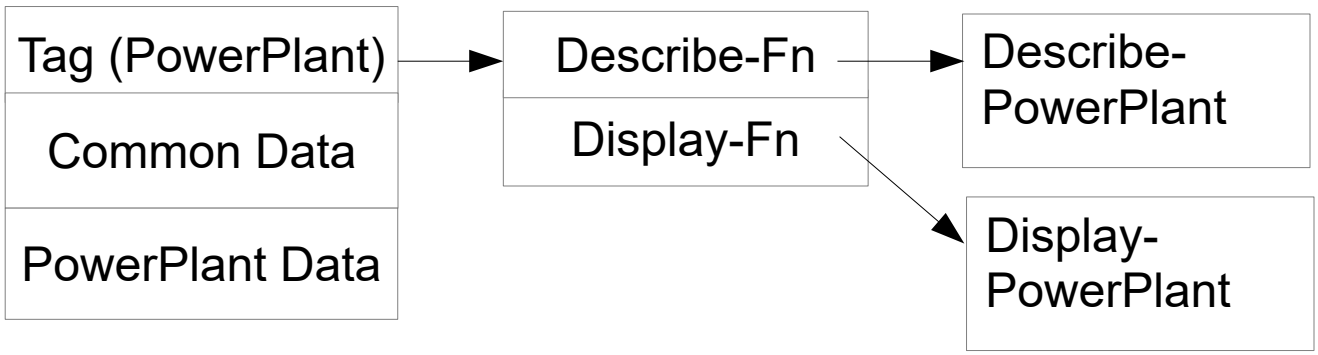


Now we only need one function pointer structure for each kind of location. In addition, certain errors become harder to make and are more obvious if we do make them.

The Last Tweak

Most of the time, there's no good reason to use a union to share structure among all direction types of location. That can waste a lot of space depending on how different the variants are. Here's the way we want to do things for PowerPlant, for example:

PowerPlant:



Residential would look similar. The C code for this might look something like the following.

Data Definitions:

```
struct LocationData;

typedef string (*describe_loc_fn) (struct LocationData *);
typedef void (*display_loc_fn) (struct LocationData *, /* etc. */);

/* VTable (Vector Table) is the vector of pointers to
   functions giving the behavior */
struct VTable {
    describe_loc_fn describe_location;
    display_loc_fn display_location;
};

struct LocationData {
    struct VTable * vtbl;

    /* Common Data */
    . . .
};

struct Vtable powerplantvtbl = {describe_powerplant, display_powerplant};

struct PowerPlant {
    struct LocationData common;

    /* Power Plant specific data */
}
```

Functions:

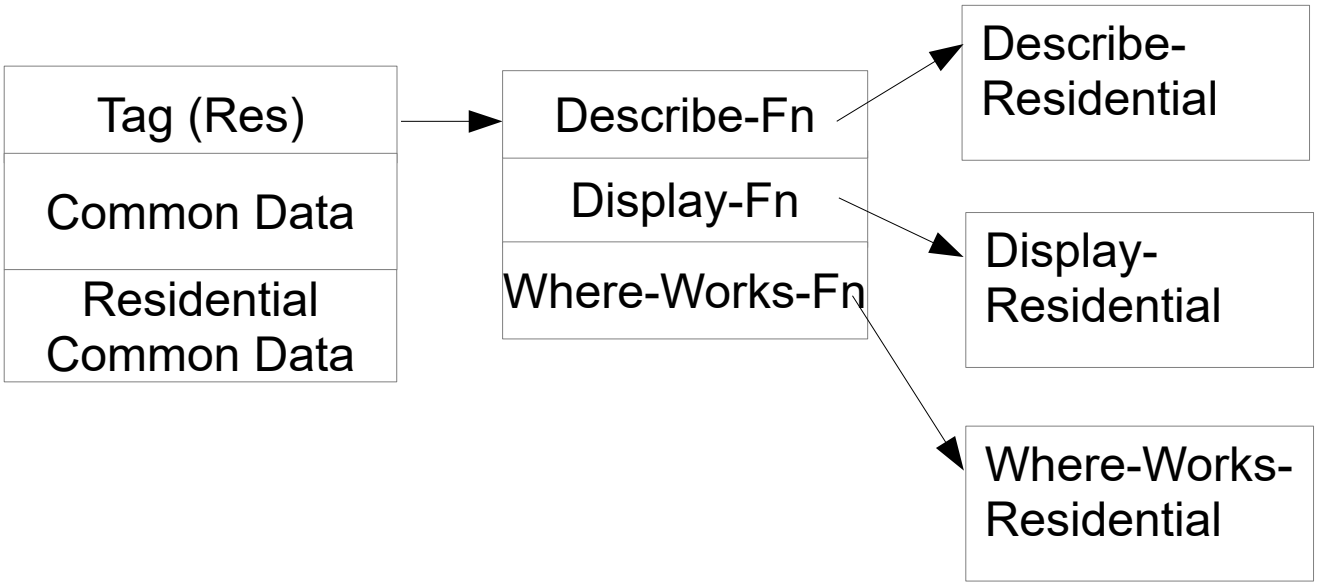
```
string describe_location(struct LocationData *loc) {
    return loc→vtbl→describe_location(loc);
}

string describe_powerplant(struct LocationData *loc) {
    struct PowerPlant *this = (struct PowerPlant *)loc;

    /* create and return description */
}
```

We are using the fact that `PowerPlant` starts with `LocationData`. The C language guarantees that I can cast pointers in the way that is done here and that it will work as expected. Note that it's an error with unknown consequences—probably bad—to pass a `LocationData` object that is not a `PowerPlant` to `describe_powerplant`.

Inheritance

Using this structure allows us to easily implement the object-oriented concept of *inheritance*. Suppose that residential zones have a third behavior: a *where do residents work* function that returns a description of where the people who live in that residential zone work. (That could be important for figuring out road usage.) That gives us structures that look like this:

Residential:

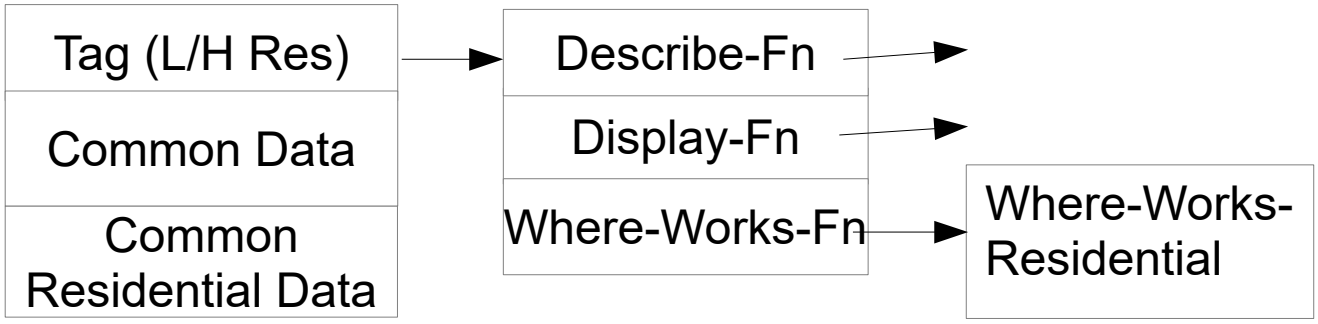


We can take advantage of the same trick we used above: The vector table is organized in such a way that the functions representing the common behavior are in the same place in memory for all kinds of locations. Even though residential zones have an additional behavior, that additional behavior does not disturb the locations of the common (describe and display) behavior. This means that the code for the functions implementing behavior don't have to check to see which kind of location it is being called with. If you want to describe a location, you use the tag to locate the vector of function pointers and the first function pointer will always point to the describe function for that kind of location.

In the figure there is only one describe function and one display function for residential zones. When we do this, we are assuming that these functions only access the data that is common among all locations, common data, or common for residential zones.

Suppose there are two different kinds of residential zones, low density and high density, with distinct functions display and describe functions for each kind of zone. There are now two different function pointer structures, one for each kind of residential zone. Those structures point to the same where works function and the correct describe and display functions for each kind of zone.

Conceptually, there is an *abstract* parent to both kinds of zones that looks like this.

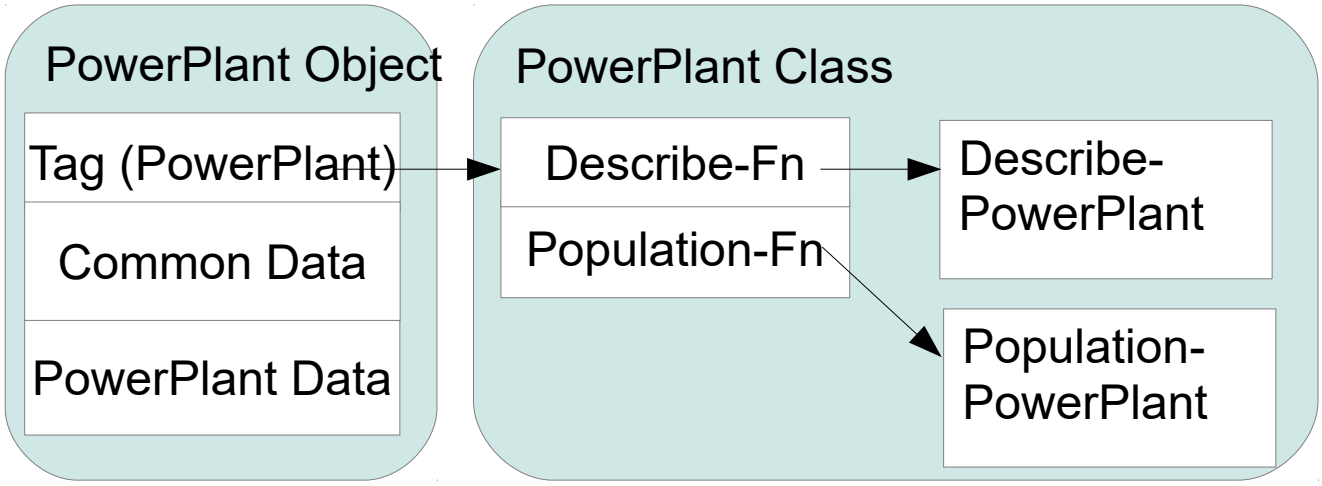


This structure is abstract in the sense that no version of this structure ever actually occurs when the program runs. However, this is the structure from the perspective of the where works function for residential zones. Note that the describe function can be called by the where works function because any *concrete* (actual) residential zone will have a describe function, even though the specific function that will be called is not known.

Object-oriented programming languages, such as Java, C++, and Python, allow us to define the abstract structure, once and then inherit that description and extend it for low-density and high-density residential objects.

Objects and Classes

Conceptually, the pictures above show how *object*s and *class*es work and how they relate to each other:



The data structure for the object contains that object's data together with a pointer to a "class" structure. The class reference can be found in the same location in all objects. The class structure references or contains the associated behavior, typically referred to as *method*s or operations for the object. All objects of the same class share this single class structure (class object?).

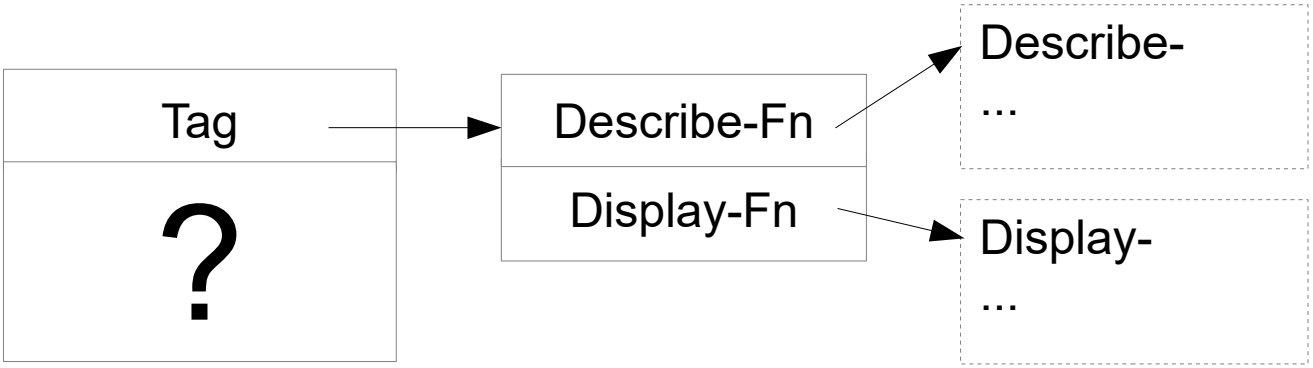
This is exactly how C++ implements objects (ignoring multiple inheritance). Note that in C++, if you change the behavior of a class, for example, by adding a new method, you have to recompile all the code using the class, because all the code using that class has

expectations about exactly where the method references are located in the class structure.

Conceptually, Java also works the same way. However, because Java handles compilation and execution differently from those languages, it uses a more complex internal model.

Interfaces and Abstract Data Types (ADTs)

If behavior is simply a set of functions that have the required signatures and correct semantics (do the right thing), you can abstract that notion to the concept of interfaces and ADTs (Abstract Data Types). In our example, locations all exhibit a common set of behavior: they have a describe function and a display function. If we abstract this, we get the following picture:



This structure can be accepted as a parameter by any function or method that expects its parameter to be a structure that has describe and power functions with the expected signatures (parameters and return type) and semantics (behavior).

What we have done is to abstract away all the details about the kinds of locations and how they're implemented. We are left with the abstract concept of a location being an object that has describe and display functions.

This idea leads to two, closely related, concepts:

- *Abstract Data Type (ADT)*. This is a mathematical and program structure concept that views objects as being defined by their behavior, that is, the operations that the object provides.
- *Interface*. This is a programming language construct, frequently used as a way to implement an ADT, that allows multiple different classes of objects to be manipulated by functions that expect objects that implement the interface.

Types versus Classes

For this class:

1. In general, ***type*** refers to the type of the object in the sense of an ADT. That is the behavior provided by the object.
2. When discussing a variable or attribute in a Java program, ***type*** refers to the declared type of the object.

3. ***Class*** refers to the actual class of an object. That is, the specific details of the data contained in the object and how the operations of the object are implemented.

For example, in the residential example, above, the type of an object could be location but the class would have to be either power plant or residential.

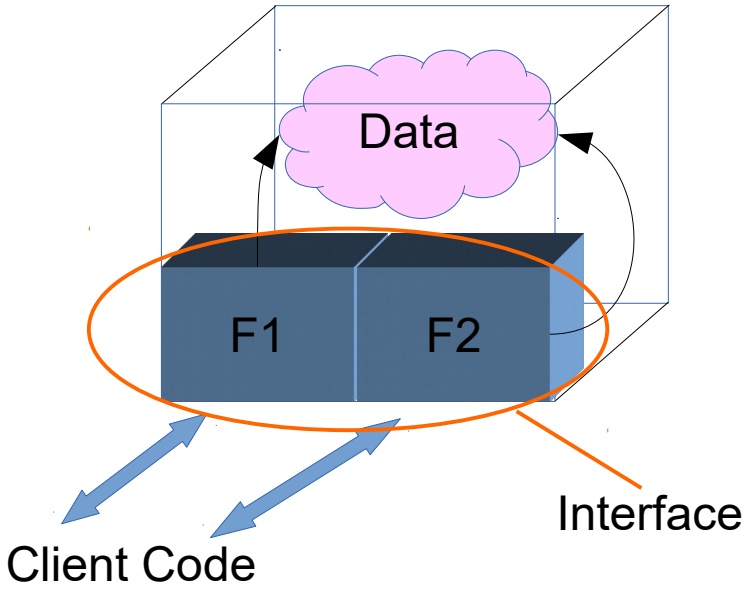
Note that a given object can only have one class but it can have multiple types.

Objects versus Structured Programming

As previously discussed, one of the problems with structured programming is that all data is exposed to the entire program. You would like to be able to control the way the data is manipulated to maintain correctness.

For example, by treating a date as an abstract data type and controlling the construction and manipulation of dates, we can avoid creating February 31, 2016 (or February 29, 2015)

Objects provide a mechanism for doing this. Objects put the data and the functions that manipulate that data together. We can think of the data as being "inside" the object. The functions (operations, methods) on the object are (or can be) the only way to access and/or manipulate the data. We say that the data is *hidden* or *encapsulated* in the object.



For example, consider a Location object in a SimCity like Game. We can think about the kinds of data that might be contained in the object. For example:

- Location in the City
- Kind of location
- How many people live or work here
- Whether the location has power

- etc.

We don't want the clients of the a given location object to manipulate this data directly. Rather, we want to add behaviors (operations, methods) that manipulate the data in ways that consistently update all the data in the object. For example, here is a possible set of behaviors:

- Describe the Location
- Update the Location based on time passing
- Provide information about the location, e.g. is it powered?, how many people live there?

Note that, for example, the number of people living in a given location is based on many factors and changes with the passage of time in the game.