Unit 2. Encapsulation

In this Chapter:

2.1 Building a Class

Sam loves Object-Oriented programming. Unfortunately, he needs to use a language that does not support classes. What can he do? Is he forced to go back to his old procedural ways? Just as he was about to give up all hope, he remembered that you can always create a class from a structure. With relief, he continues with his project.

Objectives

By the end of this chapter, you will be able to:

- See the relationship between a structure and a class
- Appreciate how simple it is to build a class from a structure
- Know what a v-table is and how to create one
- See the necessity of the this pointer

Prerequisites

Before reading this chapter, please make sure you are able to:

- Create and use a structure (Chapter 1.3)
- Create and use a pointer to a function (Chapter 1.5)
- Understand the components of a UML class diagram and know what each component is used for (Chapter 2.0)

What is a v-table and why you should care

While the value of classes in programming problems is readily apparent, it is less obvious how to create one. The answer, it turns out, is actually quite simple. While a structure consists only of member variables, a class consists of member variables and member functions. If we can find a way to add functions to a structure, we would have a class. The most straightforward way to do this is through function pointers.

A virtual function table, also known as a v-table, is a structure containing function pointers to all the methods in a class. By adding a v-table as a member variable to a structure, the structure becomes a class.

This brings us to the "why do I care?" part of the chapter. Most programming languages such as C++ hid the v-table from the programmer. A programmer could go through his entire career without directly seeing or using a v-table. Why, then, would we want to learn about v-tables? The answer is subtle. All the major concepts in Object-Oriented programming (encapsulation, inheritance, and polymorphism) are directly or indirectly influenced by the v-table. If these concepts were easy to understand, then there would be little need to discuss the v-table. Unfortunately they are not! Many of their behaviors seem mysterious and are difficult to predict. However, by understanding how the v-table works behind the scenes, the mystery and unpredictability disappear.

Building a class with function pointers

Perhaps the best way to explain how to build a class is to do so by example. Consider the Card class from Chapter 2.0:

Card
card
set
getRank
getSuit

We will start by implementing the three methods as though they were not members of the class. Since all three methods access the member variable card, each method must take a Card as a parameter. By convention, we pass this object by-pointer rather than by reference as we normally would:

```
void set(Card * pThis, int iSuit, int iRank) // we need to pass pThis as well a
{ // and iRank because pThis changes
    pThis->card = iRank + iSuit * 13;
}
int getRank(const Card * pThis) // pThis is const because getRank does not
{ // change pThis
    return pThis->card % 13;
}
int getSuit(const Card * pThis) // here too pThis is const because getSuit
{ // does not change pThis
    return pThis->card / 13;
}
```

Now, to add these three member functions to Card, we need to add three function pointers:

```
struct Card
{
    int card;
    void (*set )( Card * pThis, int iSuit, int iRank); // wow the function
    int (*getRank)(const Card * pThis); // pointer syntax
    int (*getSuit)(const Card * pThis); // is ugly!
};
```

The final step is to instantiate a card object. This means it will be necessary to initialize all the member variables. Unfortunately, this is a bit tedious:

```
{
    Card cardAce;
    cardAce.set = &set; // this is tedious. Every time we want to
    cardAce.getRank = &getRank; // instantiate a card object, we need to
    cardAce.getSuit = &getSuit; // hook up all these function pointers!
    cardAce.set(&cardAce, 3, 0); // calling the function is sure easy!
}
```

Building a class with v-tables

The most tedious thing about building a class with function pointers is that, with every instantiated object, it is necessary to individually hook up all the function pointers. This can be quite a pain if the class has dozens of methods. We can alleviate much of this pain by bundling all the function pointers into a single structure. We call this structure a v-table.

A virtual method table or v-table is a list or table of methods relating to a given object (the "virtual" adjective will be explained later in the semester). By tradition, the name for this table in the data-structure is __vtptr. Back to our Card example, the v-table might be:

```cpp
struct VTableCard
{
    void (*set )( Card * pThis, int iSuit, int iRank); // exactly the same
    int (*getRank)(const Card * pThis); // function pointers
    int (*getSuit)(const Card * pThis); // as before
};
```

From here, we can create a single global instance of vTableCard to which all Card objects will point:

```cpp
const VTableCard V_TABLE_CARD = // global const which all Card objects will poi
    {
        &set, &getRank, &getSuit // here we hook up all the function pointers o
    }; // when we instantiate V_TABLE_CARD
```

Now the definition of the Card is much easier:

```cpp
struct Card
{
    int card; // the single member variable
    const VTableCard *__vtptr; // all the member functions are here!
};
```

So how does this change the use of the Card class? Well, instantiating a Card object becomes much easier but the syntax for accessing the member functions is much more complex:

```cpp
{
    Card cardAce;
    cardAce.__vtptr = &V_TABLE_CARD; // with this one line, all the function po
                                     // are connected in a single assembly comma
    cardAce.__vtptr->set(&cardAce, 3, 0);
}
```

Note that while __vtptr is a member variable of Card and, as such, requires the dot operator. However, _vtptr itself is a pointer. It is necessary to either dereference it with the dereference operator * or use the arrow operator -> when accessing its member variables.

📋 **Sam's Corner**

On the surface, the VTable implementation of an object does not appear to have a clear advantage to the simpler function pointer. Though it is smaller (only one copy of the function pointer table is

required), it is slower to reference and much more inconvenient to use. These disadvantages are mitigated with classes where the syntax is greatly simplified and the compiler handles most of the housekeeping details.

The full code for this example is available at 2-1-cardVTable.html: /home/cs165/examples/2-1-cardVTable.cpp

Example 2.1 - Text with Function Pointer

This first example will demonstrate how to implement a class using function pointers. This class will have two member variables and two methods.

Create a class to implement the notion of text. Use the following UML class diagram as a guide:

Text
data
length
copy
isEqual

The first step is to define the Text class with two member variables and two function pointers:

```cpp
struct Text
{
    char data[256];
    int length;
    void (*copy )( Text * pDest, const Text * pSrc );
    bool (*isEqual)(const Text * pText1, const Text * pText2);
};
```

Note that the function pointers are not initialized. Each time a new instance of Text is initialized, the copy() and isEqual() function pointers will need to be assigned to the appropriate functions:

```cpp
{
    // create two text objects
    Text text1;
    text1.copy = &textCopy;
    text1.isEqual = &textIsEqual;
    Text text2;
    text2.copy = &textCopy;
    text2.isEqual = &textIsEqual;
    // initialize text1
    cin >> text1.data;
    text1.length = strlen(text1.data);
    // copy into text2
    text2.copy(&text2, &text1);
    assert(text2.isEqual(&text2, &text1));
}
```

Observe how the member functions are referenced in exactly the same way one would reference the member functions of cout.getline() and fin. fail(): with the dot operator.

As a challenge, add the method prompt to the Text class.

```
void prompt(Text & pThis, const char * message);
```

Example 2.1 - Text with V-Table

This second example will demonstrate how to implement a class using V Tables. This class will have two member variables and two methods.

This problem will implement the same UML as the previous example. The output is:

```
Text 1: Join the Dark Side
Text 2: Join the Dark Side
They are the same!
```

...and...

```
Text 1: No, Luke, I am your father!
Text 2: Nooooooooooooo!
They are not the same!
```

We start by building a struct called Text (which contains the variables, in other words, the nouns):

```
struct Text
{
    char data[MAX];
    int length;
    const VTableText * __vtptr;
};
```

The body of vTableText (which contained the functions, or in other words, the verbs) is:

```
struct VTableText
{
    void (*copy ){ Text * pDest, const Text * pSrc );
    bool (*isEqual)(const Text * pText1, const Text * pText2);
};
```

Now this will only contain the member functions that will be in Text. Now to initialize VTableText:

```
const VTableText V_TABLE_TEXT =
{
    &textCopy,
    &textIsEqual
};
```

To instantiate a Text object, we need to initialize the __vtptr member variable:

```
{
    Text text1;
    text1.pVTable = &V_TABLE_TEXT;
}
```

As a challenge, add the method prompt to the Text class. void prompt(Text & pThis, const char * message);

Problem 1 - 3

Problem 1 through 3 will be done using the function pointer method of implementing a class. Giving the following class definition:
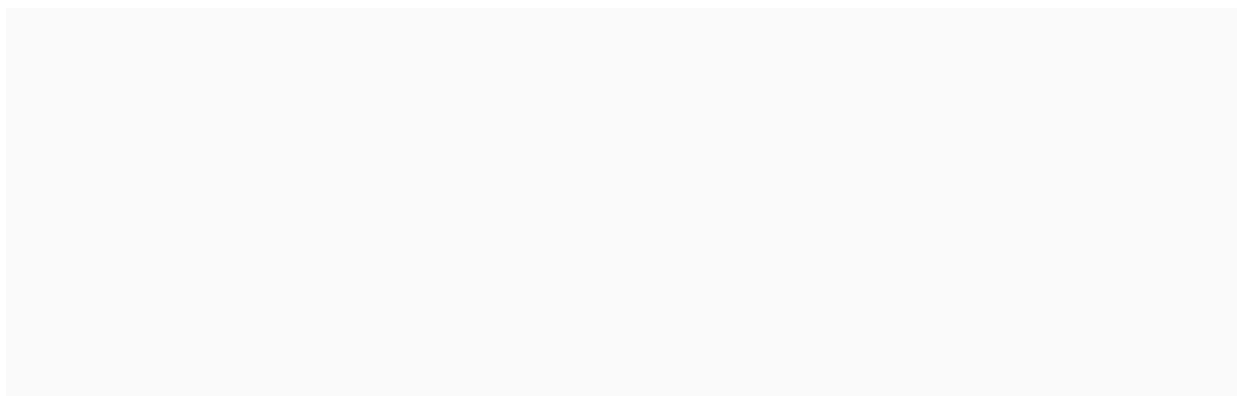
Position
row
col
set
display

1. Define the Position class using a structure and function pointers.

2. Implement the `set()` and display() methods. Do not forget to pass the pThis pointer as the first parameter!

3. Create a variable of type Position, set the value to $(4, 3)$, and display the results. Assume that there are functions defined called set(Position *, int, int) and display(const Position *).

Problem 4 - 5

Problem 4 and 5 will be done using the v-table method of implementing a class. Giving the following class definition:

Position
row
col
set
display

4. Define the Position class using a structure and the vTablePosition structure.

5. Create a variable of type Position, set the value to $(4, 3)$, and display the results. Assume there the following global constant variable is defined:

```
VTablePosition V_TABLE_POSITION =
{
    &set;
    &display;
};
```