General Setup of a File Working With Opaque Objects

Header Files

- **object header file** - use Date object as an example
  - Since the file is for a Date object, call the file date.h
  - The things you should include in the object header file are:
    - **Header guards:** #ifndef and #endif statements
    - any other necessary header files like status.h
    - typedef for the opaque object handle
    - function declarations
  - *Naming conventions*
    - The structure is named Date so the handle should be DATE
    - The file is named date.h so the macro for the header guards should be DATE_H
    - "hDate" - the *h* stands for **handle**
    - "phDate" - the *ph* stands for **pointer to a handle**
    - function names begin with the name of the object
  - *Putting everything together*

```
#ifndef DATE_H

#include "status.h"

typedef void* DATE;

DATE date_init_default(void);
Status date_set_day(DATE hDate, int new_day);
Status date_set_month(DATE hDate, int new_month);
Status date_set_year(DATE hDate, int new_year);
void date_destroy(DATE* phDate);

#endif
```

- **status.h header file**
  - Declare the Status and Boolean enumerated types with header guards

```
#ifndef STATUS_H

// most concise method, put the enum declaration and typedef all in one line
typedef enum status { FAILURE, SUCCESS } Status;
typedef enum boolean { FALSE, TRUE } Boolean;

#endif
```

  - It is important that FAILURE and FALSE both come before SUCCESS and TRUE. The first item in an enumerated type has a default value of 0, and each following item gets incremented by 1. By having FAILURE and FALSE be first, this sets their value to 0 (false) and SUCCESS and TRUE are true (1). Remember in C, the number 0 is false and everything else is true.

Object implementation file

- Since the file is for a Date object, call the file date.c
- Include the object header file date.h. Any other header files already included in date.h (i.e. status.h) do not need to be included here since they are already included in date.h
- Include function definitions - note how the functions use the handle DATE and not the regular pointer Date* in the formal parameters
- Include the structures (one or more depending on the object - one in this example)
    - Note the naming convention. The structure is called "struct date" so the typedef names it Date where the first letter is capitalized.
- Include any necessary standard C libraries like <stdio.h>, <stdlib.h> etc.
- *Putting everything together*

```
#include <stdio.h>
#include <stdlib.h>
#include "date.h"

typedef struct date {
    int day;
    int month;
    int year;
} Date;

DATE date_init_default(void) {
    // code here
}

Status date_set_day(DATE hDate, int new_day) {
    // code here
}

Status date_set_month(DATE hDate, int new_month) {
    // code here
}

Status date_set_year(DATE hDate, int new_year) {
    // code here
}

void date_destroy(DATE* phDate) {
    // code here
}
```


Main

- include object header file date.h and any necessary standard C libraries. Do not include the object implementation file date.c
- You do not need to include the status header file since it is already included in the object header file date.h
- *Putting everything together*

```
#include <stdio.h>
#include <stdlib.h>
#include "date.h"

int main(int argc, char* argv[]) {
    DATE hDate = date_init_default( );
    date_set_day(hDate, 5);
    // ….
    // ….
    date_destroy(&hDate);
    return 0;
}
```


How Handles / Opaque Objects Protect Data

- Objects are structures and you typically carry them around in pointers. This is because malloc returns a pointer, so if you want to use malloc to create an object, you have to store it in a pointer.

```
Date date = malloc(sizeof(Date));        // invalid
Date* pDate = malloc(sizeof(Date));      // valid
```

- The point of objects is to be able to hold a lot of information about one thing in a single object.

```
// all separate variables, not related to        // variables are all in one object for a date,
// each other.                                    // related to each other via the object
int day = 1;                                      Date date;
int month = 1;                                    date.day = 1;
int year = 1970;                                  date.month = 1;
                                                  date.year = 1970
```

- An opaque object is still an object, it's just a special type of object that is carried around in a void pointer (void*) instead of a regular pointer. This is explained in more detail below, and there is a demonstration on the next page.
  - A **handle** is a void pointer to an object (void*). Another term for a handle is an **opaque object**. A handle to an object cannot access the data inside of the object because it doesn't know the type of the object. So, a handle is a pointer to an object of some unknown type. A handle knows the address of a building, but it doesn't know what type of building it is nor does it have the keys to get inside the building.

```
DATE hDate;
hDate->day = 5;  // invalid
```

  - A **raw / regular** pointer can access the data inside of the object since it knows the type of the object. A regular pointer knows the address of a building AND it knows what type of building it is AND it has the keys to get inside the building.

```
Date* pDate;
pDate->day = 5;  // valid
```

  - So, the handle and regular pointer are similar in that they both hold the address of the object. They are different in that only the regular pointer knows the type of the object and can access the object's data.
- *How handles protect data:*
  - The first line of defense is that a handle (DATE) holds the address of the object but it cannot access the object's components.
  - The second line of defense is that any and all structures (Date in this example) are put in the .c implementation file (date.c) and not the .h header file (date.h). Since we don't use an include directive in main for date.c, the structure Date is not recognized by main. Therefore, you could not create a regular Date pointer (Date*) in main which would allow you to access the data items and potentially set them to invalid values.
  - What both these lines of defense do is create a program where you could never intentionally or accidentally set an object's data item to an invalid value like setting a day to a negative number in the Date example.

- In summary:
  - 1) you can only use the handle DATE and not a regular pointer Date*.
  - 2) Since you can't access and therefore modify data items with a DATE handle, you must write functions to modify the data.
  - 3) If you write the functions so that there is error checking to prevent a data item from getting set to an invalid value, then you have created a program where there is quite literally no way to set a data item to an invalid value either intentionally or by accident.
    - For the Date object, error checking would mean that the day's range varies by month (1 - 30, 1 - 31, 1 - 28, 1 - 29 if you account for leap year), the month's range is always 1 - 12, and the year must be a positive number.

```
#include "date.h"
// note how there is no #include "date.c".

int main(int argc, char* argv[ ]) {
    // line of defense 1
    DATE hDate = date_init_default( );
    hDate->day = -5;                    // invalid, can't access data with the handle DATE

    // line of defense 2 - part 1
    Date* pDate = (Date*)hDate;   // invalid, main doesn't recognize Date.
    pDate->day = -5;

    // line of defense 2 - part 2
    ((Date*)hDate)->day = -5;       // invalid, main doesn't recognize Date

    // valid, and the date will remain unchanged since you wrote the function to
    // ignore negative values. After the function call ends, the object is still in a valid state.
    date_set_day(hDate, -5)

    return 0;
}
```


More On Interface Functions

- **Interface functions**: these are the functions that are declared in date.h and defined in date.c. They allow you to use objects and modify their data in a safe and secure way.
- *Casting to the known type:* if you need to access the data in the object within a function, then you must **cast to the known type**. This is when you copy the address of the object - currently stored in the handle - into a regular pointer. You can now access the object's data via the regular pointer. The typecast "(Date*)" on "(Date*)hDate" is optional.

```
Status date_set_day(DATE hDate, int new_day) {
    Date* pDate = (Date*)hDate;  // cast to known type
    if (new_day is invalid)
        return FAILURE;
    pDate->day = new_day;        // valid because pDate is not a handle
    return SUCCESS;
}
```

- *Initializer and Destroyer:* all object interfaces should have a default initializer and destroyer. Additional customized initializers can be made (like initializing with a specific day, month, and year), but at a minimum you should always have a default initializer that sets the data values to some arbitrary but valid values. The typecasts are again optional.
  ** **Note:** the way malloc is done here is not considered best practice - it's taught this way since it's a bit easier to understand at first. The best practice for malloc for real life C programs can be seen in "ProperUseOfMalloc.c" file **

Initializer

```
DATE date_init_default(void) {
    Date* pDate = (Date*)malloc(sizeof(Date));      // typecast optional
    if (pDate != NULL) {
        pDate->day = 1;             // any values can be used here. Just make sure
        pDate->month = 1;           // they're valid for day, month, and year
        pDate->year = 1970;
    }
    return (DATE)pDate;             // typecast optional
}
```

Destroyer

```
void  destroy_object(DATE *phDate);
```

Note how the destroyer uses **pass by reference** for the handle by taking **a pointer to a handle**. Since a handle is itself a pointer, this means phDate is a pointer to a pointer to an object. This is done to avoid a **dangling pointer**, also called a **wild pointer.** This is when an object gets destroyed, but its pointer (handle in this case) isn't set to NULL afterwards. By using pass by reference for the handle, the function can set the handle to NULL thereby avoiding the creation dangling pointer.

```
// case 1 - incorrect way
void date_destroy(DATE hDate) {
    Date* pDate = (Date*)hDate;
    free(pDate);
    pDate = NULL;
    // or
    hDate = NULL;
}
// in main
DATE hDate = date_init_default( );
date_destroy(hDate);
// hDate has not been set to NULL and
// is a dangling pointer
```

```
// case 2 - correct way
void date_destroy(DATE* phDate) {
    Date* pDate = (Date*)*phDate;
    free(pDate);
    *phDate = NULL;
}

// in main - (note the &)
DATE hDate = date_init_default( );
date_destroy(&hDate);
// hDate has been set to NULL.
```

Case 1 - object's memory was freed successfully, but hDate is not set to NULL. pDate and hDate are local variables so by setting them to NULL, you are not setting hDate to NULL.

Case 2 - object's memory was freed successfully and hDate is set to NULL because you used pass by reference. The line "*phDate = NULL;" sets the handle hDate back in main to NULL.

- Most of the time (not always), the destroy function is the only one that uses pass by reference in a formal parameter (DATE*). All others just use a regular handle to the object (DATE)

- *When to use Status vs. Boolean:*
  - Boolean: Use this as the return value of a function that checks for some true / false condition. The most common example of its use is when writing a function to check if a particular data structure is empty or not:

```
// return TRUE if the month is July. Else return FALSE
Boolean date_month_is_july(DATE hDate);
```

  - Status: The utility of Status is a little more general than Boolean since a number of situations could have outcomes that are considered SUCCESS or FAILURE. The two most common examples are memory allocation failure, and attempting to do an operation that is invalid due to the object's current state.

```
// 1) memory allocation failure
// adding to the Vector could require a resize operation. If the resize
// operation fails, return FAILURE. Else, return SUCCESS.
Status vector_push_back(VECTOR hVector, int newItem);

// 2) attempted operation is invalid due to the object's current state.
// returns SUCCESS if new_day is a valid day given the month i.e.
// if the month is July, valid days are [0, 31]. Else returns FAILURE.
Status date_set_day(DATE hDay, int new_day);
```

