Lab 1 - Introduction to OOP (due on week 3)

Icebreaker

- First, form groups of 3 or 4 with people near you.
    - Introduce yourself:
        - Name
        - Class standing
        - 1+ major-related interest
        - 1+ hobby
- Finally, please receive your group number from the TA.

Checkpoint (Review if you don't know these:)

- malloc() and free()
- Pointer
- Struct
- Source Code ⇒ Compile ⇒ Link workflow for compiling C programs

If you're using Wiliki

- Download FileZilla if you're using wiliki
- Connect to URL:
    - `sftp://wiliki.eng.hawaii.edu`

The Badly Written Bank Account

- In Lecture 2 from the old EE205 lecture notes, *An Example Object* gives the example of a "bank account object," as seen in Figure 1.

An Example Object

- A bank account object has:
    - A set of services (also called behaviors or operations):
        - Depositing money
        - Withdrawing money
        - Getting the balance
        - Displaying transactions.
    - Some state needed to carry out those behaviors (also called instance data or local data):
        - A balance
        - A list of transactions.

Adapted from lecture note by Alex Quilici

Figure 1 - *An Example Object* from the old Lecture 2.

- **How would you organize such a program?** Given your knowledge from EE160 or other classes, what would be your optimal design for such a program? Discuss this among your group, and across groups if you want.

Obtaining the Lab Code

- Since you will be working primarily in a POSIX-like environment, you will be receiving your files in a double-layered file format: **.tar.gz**.
  - **.tar** means that a "(t)ape (ar)chive" was first created using the **tar** command
  - **.gz** means that the file was then compressed using the **gzip** command
- To compress a file or directory:
  - `tar czf <output-file> <input-file-or-directory>`
- To uncompress the **.tar.gz** file:
  - `tar xf <input-file>`
- Henceforth, you will be receiving and submitting code in this format.
- On Windows, perhaps try a program like 7zip for dealing with **.tar.gz** files.
- **Download the ee205-lab1.tar.gz file.** You'll notice upon extraction that the project/ directory is set up in a very peculiar manner. It is inspired by the directory structure of a few open-source projects. You will be using this project structure for the rest of the semester. Do not deviate from this.
- **Next, you will examine an example C program that is a badly written implementation of a bank account.** It is the program contained in the file `badly-written-bank-account.c`
  - 1. Are some parts of the program duplicated needlessly?
  - 2. Could you easily insert this code into a larger program and use a bank account as a subsystem within that larger program? Why or why not?
  - 3. Where is the bug in the program? Was it hard or easy to find? What made it hard or easy?
    - The two commented-out lines at the bottom of the file need to replace a certain two lines in the program. Try to compile, test, and run the program to see where the bug is.
- **3 different groups will then share their answer to question 1, 2, and 3**

What is an object?

- A thing that has attributes.
- Something that can do things.

Why do I care?

- When you wrote code in C, it most likely started with you worrying about how to setup a program so that the data and functions were two separate things that came together to make a program that *just worked*.
- If my guess is right, your organization and clarity in your code is most likely not perfect.

- In software, many different people will end up reading your code. **Working code is just a minimum -- clear code is desired.**
- **So: how can you organize your code to make things more readable?**

Enter: Object-Oriented Programming (OOP)

- The idea that "programs are made up of procedures" was popular in the past until a question popped up:
    - *What if I could model programs as interactions between self-supporting actors?*
- In other words: *What if I could split up my program into self-sufficient, self-encapsulated parts that tried to model real-world **objects**?*
- Imagine: A **Car** as an object, that has functions/operations it can perform:
    - Opening doors ⇒ `Car_open()`
    - Cleaning the windows ⇒ `Car_clean_window()`
    - Filling up gas ⇒ `Car_fill_gas()`
- Now, when you write code with this **Car** "object," you'll be thinking:
    - *What can I do with the car?*
      Instead of:
    - *What variables or status flags do I need to be concerned about when calling a function?*
- **Object-oriented programming** is concerned about designing programs in terms of objects instead of functions and variables. You'll be trying to make most things into objects.
- And, in general:
    - Since objects are self-sufficient and try to hide data that's only important to themselves -- less variables to worry about.
    - Since objects define only what they can do -- bugs will only be in how the object interacts with other objects
- **Success!** Your code is now less complex and more easier to read by using object-oriented design over using functions/procedures (which is called structured programming).

Mimicking Objects in C

- C is not an object-oriented language -- it has no language feature that directly supports OOP. C is known as an imperative, structured programming language.
    - "I**mperative**" means that the way you describe programs is by telling what actions it should take to accomplish a task.
    - "**Structured**" because C's primary means of abstraction uses functions over gotos or objects.
- However, that doesn't mean you can't program in an OOP style in C!
    - One of the most important parts of OOP is called "**encapsulation**." Essentially, objects should encapsulate and isolate all the data that concerns itself, and actively try to hide its own data. This is also called **information hiding.**

- Note that despite C being classified as an imperative language, you can perform encapsulation in the language. You can use a feature called opaque pointers: pointers to structs whose member variables is not visible to a user file (but may be eventually linked in with a different source file)
  - **You will attempt to demonstrate this for this lab.**

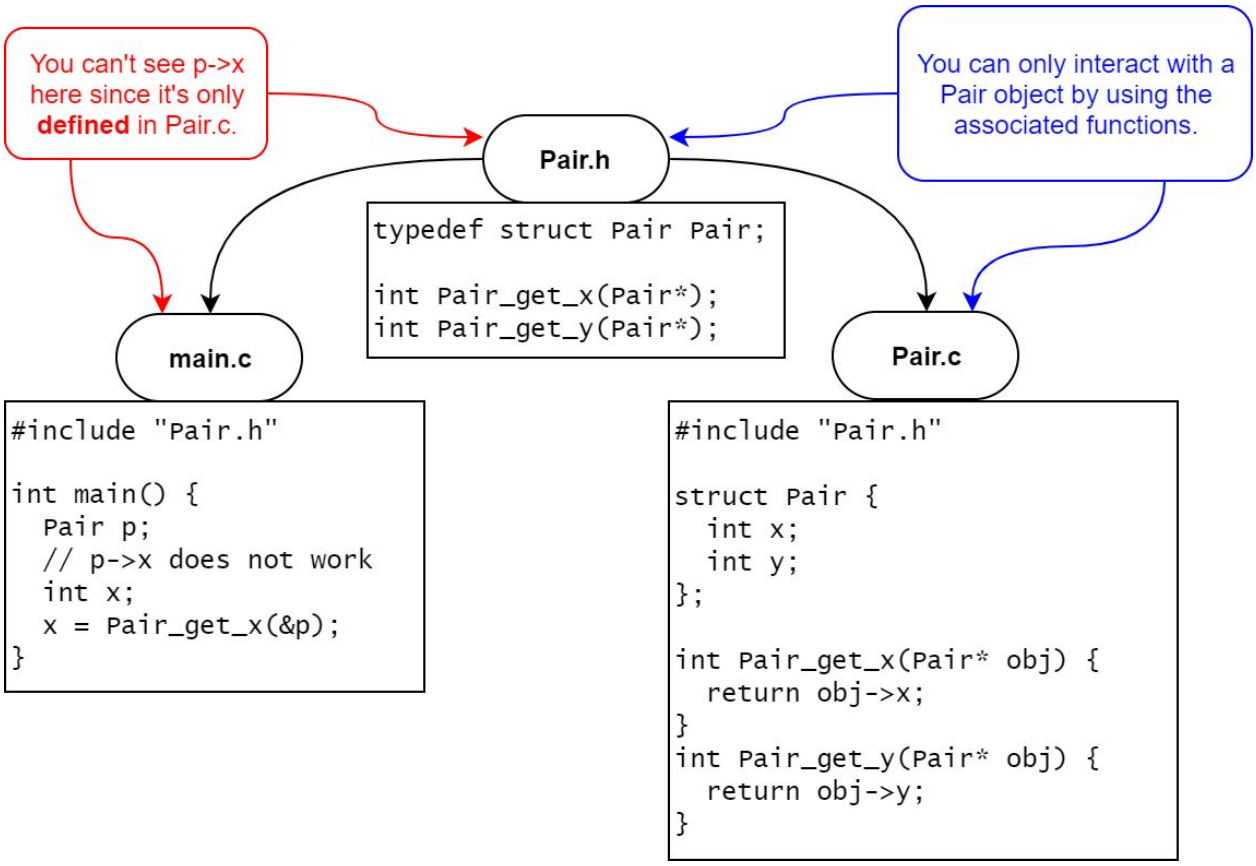

Figure 2 - The **Pair** struct is turned into a pseudo-object by hiding its internal variables (information hiding/**encapsulation** of object state) and forcing users of the object (**`main.c`**) to interact with it by only using the associated functions.

Task

- To refresh yourself on C, as well as to introduce basic object-oriented programming, your tasks is to implement the functions from `BankAccount.h` in `BankAccount.c`.
- The BankAccount should have functions paired with it to perform:
    - **Initialization** of a BankAccount struct, allocated with **malloc()**, returning a pointer to the newly created struct. It should initialize the internal balance to 0, as well as any other member variables.
    - **Destruction** of a BankAccount struct, with the memory freed with **free()**
    - **Depositing** of money into a BankAccount
    - **Withdrawal** of money from a BankAccount
    - **Getting**
        - the balance of a BankAccount
        - the last withdrawal amount of a BankAccount
        - the last deposit amount of a BankAccount
    - At no point should your functions be returning a value of the type **BankAccount**. (Remember, the hypothetical types **A** and **A*** are two different types, since one is A and the other is "pointer to A")
- Your BankAccount struct should be implemented in 2 files:
    - `BankAccount.h (hint: do not modify this file at all)`
    - **`BankAccount.c (only touch this source file for the remainder of the lab)`**
- **Do not touch `main.c`. It is there to provide a contrasting example that's better written than `fixed-badly-written-bank-account.c`.**
- To test to see whether your program matches the behavior of **fixed-badly-written-bank-account.c**:
    - You can use files to redirect input and output to files:
        - **`gcc BankAccount.c`**
        - **`./a.out < input.txt > output1.txt`**
        - **`./b.out < input.txt > output2.txt`**
    - Try to compare the output of your own executable and the **badly-written-bank-account.c** program by using the **diff** utility program
        - **`diff -u output1.txt output2.txt`**
            - **This command will not print out anything once the `output.txt` files are EXACTLY the same.**

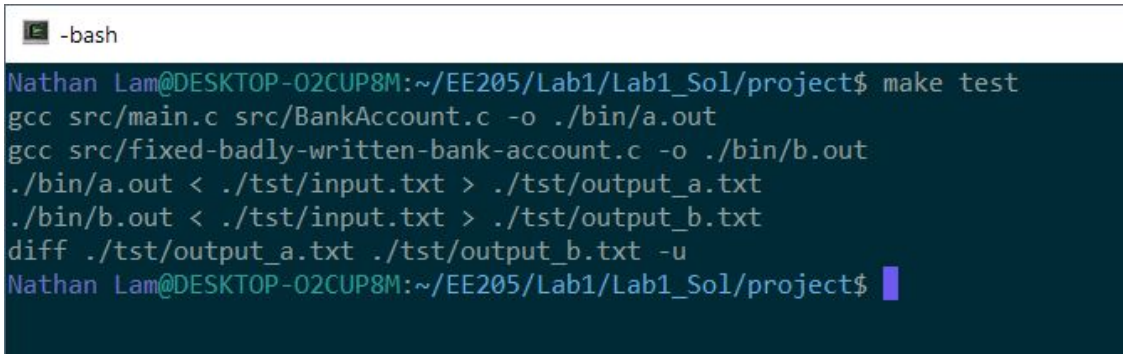

Figure 3 - You can also use the command **make test** that invokes the Makefile to build the programs, test them with the same input file, and compare the outputs.

- Your final executable should have the filename **a.out**.
- For full credit, the output from your **./a.out** executable should match that of the **./b.out** executable (which is created from **fixed-badly-written-bank-account.c**) if they receive the same input.
- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**

Submission

- In a file called **team.txt**, write your group number alone on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, “da Silva”), you may keep the whitespace.
- Place your source code files into a directory called **project**, using the following directory structure:
    - **project/**
        - **Makefile**
        - **docs/** (stands for “documentation”)
            - ***team.txt***
        - **src/** (stands for “source”)
            - **<put your source files here>**
        - **bin/** (stands for “binary”)
            - **<when Makefiles are introduced, your executables should appear here after invoking them>**
            - When submitting, NO EXECUTABLES should be in this directory. They should be able to be built by running the “make” command once you have created a Makefile.
        - **tst/** (stands for “test”)
            - **<any testing-related files will appear here>** (but not yet)
- Archive and compress your **project** directory using the command:

    - `tar czf name_of_archive_file.tar.gz project/`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file under your group's entry on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.
- For your convenience, we have provided an example Makefile that will help you build your project already.

Pro tip: use "Tab" to auto-complete command and filenames.