Object Oriented Programming and C++

Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

Problems in C

Structure in C

Structure bundles variables together.

Define a structure "student"

```
struct student{
    int ID;
    char *name;
    int overall_grade;
};
```

Declare a structure variable and initialize it:

```
struct student song;
song.ID = 1024;
song.name = "song liu";
song.overall_grade = 70;
```

It works great! However, I decide to record more detailed students info for individual programmes at a later day.

Two New Structures

Say, I define a struct called `CS_student`.

```
struct CSstudent{
    int ID;
    char *name;
    int overall_grade;
    int programming_grade;
};
```

and I also define a struct called `Mathstudent`.

```
struct Mathstudent{
    int ID;
    char *name;
    int overall_grade;
    int calculas_grade;
};
```

Problem 1: Redundancy

- There are a lot of repetitions in these three definitions!
- Repetitions <=> Code is poorly reused!
- Repetitions <=> Confusion!

```
struct Lawstudent{
    int ID;
    char *name;
    int overall_score; // you are not carefully
    // following your naming paradigm.
    int law_grade;
};
```

-
  - The user of your code gets confused: is the `overall_score` the same thing as the `overall_grade`?

Problem 2: Type Hierarchy

- The definition does not reflect that `CSstudent` and `Mathstudent` are sub-types of `student`.
- Imagine I have a function:

```
int print_overall_grade(struct student s){
    printf("%d\n",s.overall_grade);
}
```

  - By human logic, you would think the following works:

```
struct CSstudent song = {...}; //initialize "song"
print_overall_grade(song); //COMPILATION ERROR!
```

  - `song` is a `CSstudent` structure. It does not match the input type in `print_overall_grade`.

Structure Pointer

```
struct student song = {...}; //initialization code
struct student *psong = &song;
song.ID = 1234;
psong->ID = 1234; //same as above.
```

- Note when you have **a structure pointer**, instead of using `.` to refer to its variables, you need to use `->`.
- Just remember, pointer uses "pointer" (`->`)...

Setting Variables in Structure

- Imagine I have a function `set_overallscore`.

```
/*set_overallscore, record student's score.
Pass by reference, do not pass by value!! */

void set_overallscore(student *ps, int score){
    //Check if score is valid or not.
    if(score <=100 && score >=0){

        ps->overall_score = score;

    }else{

        printf("Invalid Score!\n");
    }
}
```

Setting Variables in Structure

- Our `set_overallscore` function works:

```c
#include <stdio.h>
... //definition of student omitted
void main()
{
    struct student song = {1, "song liu", 0};
    set_overallscore(&song, -2);
    //prints out "invalid score!"
    printf("%d\n", song.overall_score);
    // prints 0
    set_overallscore(&song, 80);
    printf("%d\n", song.overall_score);
    // prints 80
}
```

Problem 3: Data Corruption

- In C, functions and data are detached.
- However, nothing prevents a irresponsible programmer from doing this:

```c
#include <stdio.h>
... //definition of student omitted
void main()
{
    struct student song = {1, "song liu", 0};
    song.overall_score = 99999;
    printf("%d\n", song.overall_score);
    // prints 99999,
    // Now, song has an invalid score !!
    // no warning message!!
}
```

- The data in `song` is now corrupted!
- **Data should only be accessed and modified by using its designated procedure**.

Minor Problem: Syntax Naturality

- Say a student changes his/her name.

  `change_name(&song, "new name");`

  or

  `song.change_name("new name");`

- The latter feels more natural and "human":
  - You can literally read your code as:
  - `song` changes his name to `"new name"`.

Problems of Structure in C

1. Code is poorly reused, which leads to redundancy and confusion.
2. Does not reflect proper hierarchies of data
3. Data and operations on data are detached.
    - Data may be corrupted by illegal access.

OOP and C++

Procedural Programming (PP)

- C is a procedural programming language.
  - Your code is divided into several procedures (functions) and you write code for each procedure.
- In the previous lab, we wrote the following functions:
  - `swap`,
  - `find_max_idx`,
  - `sort`,
  - `print_array`.
- Since the lab splits into two subtasks: `sort` and `print`. `sort` itself contains smaller tasks: `find` the maximum idx, and `swap` elements.
  - We defined **a function for each task**.

Object Oriented Programming (OOP)

- In OOP, your code is divided into small parts called **objects**.
  - These parts can have hierarchies reflecting the real-world relationship between objects.
  - If an object is a `CSstudent`, then it is a `student`.
  - Preserving hierarchies leading to a better reusability of your code.
- Objects contain data **as well as procedures that operates on the data**.
  - Solves the "data-operation detachment" issue.
  - The **procedures** in an object are called "**methods**".
  - The **data** in an object are called "**fields**".

C++

- C++ is an enhancement of C, that allows OOP.
- C++ is a superset of C.
  - C++ contains all language features in C and additional features for OOP.
  - Thus, a valid C program is also a valid C++ program, but not vice versa.

```
#include <stdio.h>
void main(){
    printf("hello world!\n").
}//A valid C++ program!
```

Cautions

- C++ is **not** a language for programming novice.
- C is simple and nimble, like a **swiss army knife**.
  - Anyone can use it.
  - If you program in a principled way, C can do everything.
- C++ is powerful and complex, like a **tank**.
  - It contains powerful features, but mostly for large scale software development.
  - Using it in smaller projects may unnecessarily complicate things (overengineering).
  - If you abuse/misuse language features in C++, your program may be less readable and performant than using just PP in C.

Compiler

- C++ code are contained in `cpp` files.
  - just like C code are contained in `c` files.
- C++ uses a different compiler: `g++`.
  - It has the same usage as `gcc`.
  - `g++ main.cpp -o main.out` compiles `main.cpp` to the executable `main.out`.

Class: A More Powerful Struct

Class

- Class is the "structure" in C++.
- It groups related variables as well as procedures together in one entity.

```
#include <stdio.h>
class student{
    int ID;
    char* name;
    int grade;
};
// you do not need typedef to create an alias!
// you can use student as a type directly.
int main(){
    student song;
}
```

  - `song` is **an object** or **instance** of class `student`.

Class

- By default, all fields (variables) in a class are private
  - You cannot access those fields.

```
student song;
song.grade = 70; //WRONG! COMPILATION ERROR
```

  - You need to manually declare fields as `public`.

```
class student{
public:
    int ID;
    char* name;
    int grade;
};
```

  -

```
student song;
song.grade = 70; //OK!
```

Methods

- Methods are functions that are "attached" to an object.

```
class student{
public:
    int ID;
    char* name;
    int grade;
    void set_grade(int score){
        if(score <= 100 && score > 0){
            grade = score;
        }
    }
    int get_grade(){
        return grade;
    }
};
```

- `set_grade` saves the score to the `grade` field.
- `get_grade` returns the `grade` field.

Methods

- Methods can be called using the "dot" notation:

```
student song;
song.set_grade(70);
printf("song's grade %d\n", song.get_grade());
//prints out 70
```

  - Just like calling a regular function, you need to feed the function with appropriate inputs.
  - In this case, the **object** `song` 's grade has been modified.

Encapsulation

- Exposing your fields as `public` variables is dangerous.
  - An irresponsible programmer can corrupt your data!
  - Recall the "student score" example.

```
student song;
song.grade = 999;
printf("song's grade %d\n", song.get_grade());
//prints out 999, which is invalid score
```

Encapsulation

- To protect your data, do

```
class student{
    int ID;
    char* name;
    int grade;

public:
    void set_grade(int score){
        if(score <= 100 && score > 0){
            grade = score;
        }
    }
    int get_grade(){
        return grade;
    }
};
```

Encapsulation

- Now, nobody can corrupt your data:

```
student song;
song.grade = 999; //WRONG! COMPILATION ERROR!
song.set_grade(999); // Invalid score,
// No change to the grade field.
```

- They can only do it in "the right way":

```
song.set_grade(80); //the field "grade" is changed.
printf("%d\n", song.get_grade());
//prints out 80
```

- Encapsulation is an important idea in OOP. It prevents irresponsible programmers from corrupting and misusing data.
    - Wikipedia page on Data Hiding.

Constructor

- In C, we can initialize a structure using `{...}` syntax.

```
student song = {1234, "song liu", 70};
```

- How to initialize fields of an object in C++?
  - There is a more principled way to initialize fields in C, called "constructor".
  - Constructor is a public method that does NOT have a return type.
  - This method has the same name as your class.

Constructor

```
class student{
    int ID;
    char* name;
    int grade;

public:
    student(int newID, char* newname, int newgrade){
        ID = newID;
        name = newname;
        //checking the validity of the grade
        if(newgrade <= 100 && newgrade > 0){
            grade = newgrade;
        }
    }
    // set_grade and get_grade are omitted ...
};
```

Constructor

Then, you can initialize an object like this

```
student song(1234, "song liu", 70);
printf("%d\n");
// prints out 70.
```

Homework 1

- Write a `matrix` class.
- Contains the following **private** fields:
  - `num_rows` : integer, stores the number of rows
  - `num_cols` : integer, stores the number of columns
  - `elements` : **integer pointer**, pointing to a contiguous memory stores a row-major matrix.

Homework 2

- Write the following **public** methods in your `matrix` class:
  - `void set_elem(int i, int j, int val)` : set the `i` , `j` -th element of the matrix to `val` .
  - `int get_elem(int i, int j)` : retrieve the `i` , `j` -th element of the matrix.
  - Both methods use **zero-based index**!!
- You must check the validity of the input indices in your methods, i.e., `i` and `j` must in between 0 to number of rows and columns minus one.
- If the indices are not valid, print out `invalid indices!` .

Homework 3 (Submit)

- Write a public method `void add(matrix B)`:
  - Suppose I have two matrix objects `A` and `B` storing matrices A and B respectively.
  - If I call:
  - `A.add(B)`,
  - It would add two matrices and store the outcome to `A`.
  - $A \leftarrow A + B$
- `add` function needs to check the dimensionality of matrix B and print out `incompatible dimension!` if the dimensions of B does not match those of A.

Homework 3 (Submit)

- Write a public method `void print()` that prints out the elements of the matrix.

Homework 3 (Submit)

- Write a constructor:
  `matrix(int nrow, int ncol, int *elem)`
- It initializes corresponding fields.
- It checks the validity of `nrow` and `ncol` before assigning them to fields.

Homework 3 (Submit)

Test your implementation with provided testing code in the `main` function.