Object Oriented Programming and C++

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

Structure in C

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;
```

Two New Structures

Say, I would like to define a struct called `CS_student`.

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

Say, I would like to define a struct called `Mathstudent`.

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

Redundancy

- There are a lot of redundancies in these two definitions!
- Redundancy <=> Code is poorly reused!
- Redundancy <=> Confusion

```
struct Lawstudent{
    int ID;
    char *name;
    int final_grade;
    int law_grade;
};
```

  - Wait, is `final_grade` the same thing as the `overall_grade`?

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(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`.

Setting Variables in Structure

- In C, data and the operations on data are **detached**.
- Imagine I have a function `set_overallscore`.

```c
/*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");
    }
}
```

Structure Pointer

- Note when you have **a structure pointer**, instead of using `.` to refer to its variables, you need to use `->`.

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

- Just remember, pointer uses "pointer" (`->`)...

Data Corruption

- 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
}
```

Data Corruption

- 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!!
}
```

- Data and operations on data should be bound together
- Data should only be accessed and modified **by using the right procedure**.

Data and Functions are naturally together

- Sometimes, it is just natural that a procedure is bound to the data it operates on.
- 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. Does not reflect proper hierarchies of data
    - Code is poorly reused, which leads to redundancy and confusion.
2. Data and operations on data are detached.
    - Data may be corrupted by illegal access.
    - This programming style is arguably less natural and less "human".

Procedural Programming (PP)

- C is a procedural programming language.
  - Your code is divided into several procedures (functions) and you write code for each procedure.
- Lab 7, we wrote the following functions:
  - `read_matrix`,
  - `matrix_multiplication`
  - `write_matrix`.
- Since the task we want to perform naturally splits into these subtasks: **Read** matrices, **multiply** them and **write** to a file.
  - We defined **a function for each verb** in the above description.

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`.
- 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 geared toward large scale software development.
  - Using it in smaller projects may unnecessarily complicate things (over engineering).
  - 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

- 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 it with appropriate inputs.
  - In this case, the **object** `song` 's grade has been modified and displayed.

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

Constructor

Then, you can initialize an object like this

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

Destructor

- C++ allows you to execute a piece of code when an object is destroyed.
- This is very useful to release some resources (e.g. Heap Memory) that have been allocated in your constructor.
    - Like constructor, destructor is a public method with no return type.
    - Syntax: `~ClassName()`

Destructor

```
class Matrix{
    int numrows;
    int numcols;
    int *elements;

public:
    Matrix(int nrows, int ncols){
        numrows = nrows;
        numcols = ncols;
        //allocating heap memory for the matrix!
        printf("creating matrix...\n");
        elements = (int*) malloc(nrows*ncols*sizeof(int));
    }
    ~Matrix(){
        //memory will be freed when
        //this matrix object is destroyed.
        printf("freeing matrix...\n");
        free(elements);
    }
};
```

Destructor

```
int main(){
    //create a 2 by 2 matrix
    Matrix m(2,2);
    // do some matrix stuff...
    printf("dong matrix stuff\n");
    return 0;
}
```

The output of the program:

```
creating matrix...
dong matrix stuff
freeing matrix...
```

Although I never explicitly called `~Matrix()`, it has been automatically called before my program exits.

Life span of an object

- The life span of an object is a complicated topic in C++.
- We only need to remember a few things:
  - When your program finishes, all the objects you have created in the **stack memory** will be automatically destroyed.
  - In the same function, objects will be destroyed in the opposite order they are created.
  - Generally speaking, an object created **in the stack memory** of a function will be destroyed when the function exits.

Creating/Deleting Objects in Heap Memory

- In C++, you can directly create objects in heap memory using the `new` keyword.
- They have to be manually destroyed using the `delete` keyword.

```
//create a matrix object in the heap memory
Matrix *pm = new Matrix(2,2);
//now, m is a pointer pointing to the matrix
//now do matrix stuff... before you go
delete pm;
//the heap memory can be released by delete
//keyword, this will trigger pm's destructor.
```

Inheritance

Consider the following `student` class:

```
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;
    }
};
```

Inheritance

- Now, let us create a `CSstudent` class.
- We want to do so without duplicating the code.
    - i.e. rewriting everything we wrote for `Student` class.
- We want all `CSstudent` objects to be recognized as a `Student` object by our program.

Child Class

- Create `CSstudent` as a child class of `Student`.

```
class CSstudent: public Student{

};
```

- Now, the `CSstudent` class has **inherited** all fields and methods of the `Student` class. It can do whatever `Student` class can do.

```
CSstudent song;
song.set_grade(70);
printf("%d\n", song.get_grade()); //prints 70.
```

- Inheritance reuses my old code for `student` class, and reduces the redundancy of my code.

Child Class

- You can define fields and methods that are **exclusive** to `CSstudent`.

```
class CSstudent: public Student{
    int programming_score;
public:
    int get_programming_score(){
        return programming_score;
    }
    void set_programming_score(int score){
        if(score <= 100 && score > 0){
            programming_score = score;
        }
    }
};
```

Child Class

- Now, in addition to all fields and methods that are already in `Student`, `CSstudent` has an extra field `programming_score` and two extra methods `get_programming_score` and `set_programming_score`.
- For example:

```
CSstudent song;
song.set_grade(70);
printf("%d\n", song.get_grade());
//prints out 70
song.set_programming_score(80);
//prints out 80.
printf("%d\n", song.get_programming_score());
```

Child Class

- Moreover, all functions that take a `student` object as an input will now take `CSstudent` as input.
    - Since the C++ knows, `CSstudent` is a `student`.
- Suppose we have a function:

```
int print_grade(student s){
    printf("%d\n",s.get_grade());
}
```

- Now we can call `print_grade` using `song` as an input:

```
CSstudent song;
song.set_grade(70);
print_grade(song);
//OK, C++ knows song is a CSstudent,
// thus is a student
//prints 70.
```

Child Class

- However, once your parent class has constructors, inheritance become rather complicated.
- We will not discuss this circumstance in this unit.
- Read here for more information about inheritance.

Conclusion

- Structure in C has some issues:
  - It can not reflect the hierarchy of data types.
  - Data and operations on data are detached.
- PP: You divide your program into sub-procedures.
- OOP: You divide your program into small "objects".
  - Objects contains "fields" and "methods".
- C++
  - It is a superset of C.

Homework 1 Problem with Structure

In lab 7, we have coded structure that contains variables `numrow, numcol and elements`.

```
struct matrix{
    int numrow;
    int numcol;
    int *elements;
};
typedef struct matrix Matrix;
```

However anyone can modify `numrow` or `numcol` after the matrix has already been initialized. Imagine:

```
Matrix A = read_matrix("A.matrix");
A.numrow = 999999; //someone is being careless...
// a disaster waiting to happen...
multiply(A, B, C)
```

Homework 1 Problem with Structure

- It is a poor design if anyone can modify your data in a way that can cause a disaster.
- `numrow` and `numcol` should be **locked** once the matrix has been initialized.
- Today, we are going to see how this can be done using C++'s encapsulation.

Homework 1 Matrix Class

Create a **class** called `Matrix`. This class has the following **private fields**:

1. `numrow`, `int` type, the number of rows
2. `numcol`, `int` type, the number of columns
3. `elements`, `int` type, a **pointer** points to an array, storing the flattened matrix.

Homework 1 Indexing

4. Write a **private heler method**

```
int idx(int i, int j)
```

   - It takes the 2D index `i, j` of the current matrix, and converts it to the linearized index.
   - For example, if the current matrix is a 10 by 2 matrix. Suppose a
     - `idx(0, 0)` returns 0
     - `idx(0, 1)` returns 1
     - `idx(1, 0)` returns 2
     - `idx(1, 1)` returns 3
     - ...
   - The function `idx` should contain only one line of code.

Homework 1 Matrix Class (submit)

Write public **methods**:

1. `void zeros(int nrow, int ncol)` : allocate heap space for a `nrow` by `ncol` matrix, and fill it wit zeros.
    - Hint: use `calloc` .
2. `void print()` : print all elements in the current matrix.
3. `void fill(int nrow, int ncol, int a[])` : allocate heap space, and fill the matrix with elements in an array `a` .
    - For example,

```
int a[] = {1,2,3,4};
Matrix M;
Matrix.fill(2,2,a);
//M now stores a matrix
//1 2
//3 4
```

Homework 1 Matrix Class (submit)

4. `int get_nrow()` : returns the number row of the current matrix.
5. `int get_ncol()` : returns the number of columns of the current matrix.
6. `void free_mem()` , releases the memory occupied by `elements` .
    - Do not forget to release all heap memory you have allocated!
7. `Matrix dot(Matrix B)` : computes the matrix multiplication between the current matrix (A) and the other matrix B. Return AB.

Homework 1 Matrix Class (submit)

- Write test code in `main`, testing your methods.
    - Create a `Matrix` **object**.
    - Fill it with some elements.
    - Print out the matrix.
    - Perform Matrix Multiplication.
    - Print out the outcome of multiplication.
- In this matrix example, we have restricted the access of `numrow` and `numcol`: Once our matrix is initialized by `fill` method, `numrow` and `numcol` are read-only, hence they are "encapsulated" by our design.