Object Oriented Programming in C++

Matteo Fasiolo

About myself

Matteo Fasiolo (matteo.fasiolo@bristol.ac.uk)

GA.17, Fry Building,

Microsoft Teams (search “matteo fasiolo”).

I have been programming mostly in R since 2006, I have authoured several R packages:

- mgcViz for visualising regression models
- qgam for non-parametric quantile regression
- mvnfast for fast computation with the multivariate normal distribution

Until spring 2024 I thought a PhD-level course on how to interface R with C++.

I do research on statistical modelling for energy applications.

Course organisation

In the second part of the unit:

- week 7 introduces C++;
- week 8 covers algorithms in C++;
- weeks 8, 9 and 10 on programming in R;
- week 11 dedicated to the 2nd coursework;
- week 12 revision.

Course content will appear on Github.

Structures in C

Structures 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 lucy;
  lucy.ID = 1024;
  lucy.name = "lucy peng";
  lucy.overall_grade = 70;
```

However, what if later on we want to record more detailed students' info?

Two New Structures

Say, I define a struct called `CS_student`.

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

and I also define a struct called `Mathstudent`.

```c
struct Mathstudent{
    int ID;
    char *name;
    int overall_grade;
    int calculus_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 following
                   // your naming convention.
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:

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

By human logic, you might think the following should work:

```c
struct CSstudent lucy = {...}; //initialize "lucy"
print_overall_grade(lucy);
```

We get a compilation error!

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

Structure Pointer

To see further problems with structures, we need to introduce structure pointers:

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

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

Or alternatively

```
(*plucy).ID = 1234;
```

while the following is **wrong**

```
*plucy.ID = 1234;
```

Setting Variables in Structure

Imagine I have a safe function for setting a student's grade:

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

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

  ps->overall_grade = grade;

 } else {

  printf("Invalid grade!\n");

 }
}
```

Our `set_overallgrade` function works:

```
#include <stdio.h>
... //definition of student omitted
void main()
{
 struct student lucy = {1, "lucy peng", 0};

 set_overallgrade(&lucy, -2);
 //prints out "invalid grade!"

 printf("%d\n", lucy.overall_grade);
 // prints 0

 set_overallgrade(&lucy, 80);
 printf("%d\n", lucy.overall_grade);
 // prints 80
}
```

Problem 3: Data Corruption

In C, functions and data are detached.

Nothing prevents someone from setting grade directly:

```
#include <stdio.h>
... //definition of student omitted
void main()
{
 struct student lucy = {1, "lucy peng", 0};

 lucy.overall_grade = 99999;
 printf("%d\n", lucy.overall_grade);
 // prints 99999 and no warning!
}
```

The data in `lucy` is now corrupted!

Data should only be accessed and modified by using a specifically designed procedure.

Problems of Structures 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 unsafe access.

Object-Oriented Programming (OOP) and C++

Object-Oriented Programming (OOP) and C++

C is a **procedural programming** (PP) language.

Your code is divided into several procedures (functions) and you write code for each procedure.

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 leads to 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>
int main(){
 printf("hello world!\n");
}
```

Exception: `void main(){ ... }` works in C but not in C++.

C vs C++: Pros and Cons

C is simple/rudimental:

- Good language to start learning how to program.
- The language is close to how a computer “thinks”.
- If you program in a principled way, C can do anything.

C++ is powerful and complex:

- It contains powerful features (e.g., OOP), needed for large scale software development.
- Using it in smaller projects may unnecessarily complicate thing.
- If you abuse/misuse language features in C++, your program may be less readable and performant than using just PP in C.

Compiling C++ Code

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 is the "structure" in C++.

It groups related variables and procedures together in one entity.

```
#include <stdio.h>
class student{
 int ID;
 const char* name; // Ignore const (or see bonus tutorial)
 int grade;
};

// you do not need typedef to create an alias!
// you can use student as a type directly.
int main(){
 student lucy;
 return 0; // NOTE in C++ you need to return an int!
}
```

`lucy` is **an object** or **instance** of class `student`.

By default, all fields (variables) in a class are private.

You cannot access those fields.

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

You need to manually declare fields as `public`.

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

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

Methods

Methods are functions that are “attached” to an object.

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

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

Methods can be called using the "dot" notation:

```
student lucy;
lucy.set_grade(70);
printf("lucy's grade %d\n", lucy.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** `lucy`'s grade has been modified.

Encapsulation

Exposing your fields as `public` variables is dangerous.

A user can corrupt your data!

Recall the "student grade" example.

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

Note: here we can access `lucy.grade` because it's public.

To protect your data, do

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

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

Now, nobody can corrupt your data:

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

They can only do it in “the right way”:

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

Encapsulation is an important idea in OOP. It prevents users from corrupting and misusing data.

Wikipedia page on Data Hiding.

Constructor

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

```
student lucy = {1234, "lucy peng", 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: not even `void`!

This method has the same name as your class.

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

 public:
 student(int newID, const 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 ...
};
```

Then, you can initialize an object like this

```
student lucy(1234, "lucy peng", 70);
printf("%d\n", lucy.get_grade());
// prints out 70.
```

To summarise, we discussed the limitations of structures in C:

1. Redundancy
2. The lack of hierarchies
3. **Data corruption**

We showed how problem 3 is solved by C++ classes:

- Keep important class fields private
- Access them via safe (public) methods

Next, we will tackle problems 1 and 2.

Homework 1

Objective is to write a `matrix` class.

See `lab_matrix_template.cpp` file for initial code.

Class should contain the **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 should add two matrices and store the outcome to `A`.
- i.e., $A \leftarrow A + B$

`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 (you can make it pretty if you want!).

Write a constructor `matrix(int nrow, int ncol, int *elem)` which:

- initializes corresponding fields.
- checks the validity of `nrow` and `ncol` before assigning them to fields.

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