More Object-Oriented Programming in C++

Matteo Fasiolo

Class Destructor

In last week's lab we have created a matrix class.

The memory containing the matrix was managed outside the class:

```
int elementsA[] = {1, 2, 3, 4, 5, 6};

int num_rows = 2;
int num_cols = 3;
matrix A(num_rows, num_cols, elementsA);
```

But how to manage memory from within a C++ class?

```
class Matrix{
 int numrows;
 int numcols;
 int *elements;
 public:
     Matrix(int nrows, int ncols){
      numrows = nrows;
      numcols = ncols;
      printf("creating matrix...\n"); //alloc heap mem
      elements = (int*) malloc(nrows*ncols*sizeof(int));
    }
    ~Matrix(){
        //free memory
        printf("freeing matrix...\n");
        free(elements);
    }
};
```

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

The printed output of the program:

```
creating matrix...
doing matrix stuff
freeing matrix...
```

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

Lifespan of an object

The lifespan 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 (last in first out).
- An object created **in the stack memory** of a function will be destroyed when the function exits unless it is the return value.
- An object created in the **heap memory** will not be destroyed until it is manually freed by the programer.

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, pm 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.
```

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.
   - Solved! Classes combines variables and functions.

We will see how **the first two problems** are solved using OOP!

Inheritance

Inheritance is a relationship you can declare between classes and is the way OOP reuses code.

Inheritance expresses "is-a" relationship between classes:

- `CSstudent` is a `student`.
- `sportscar` is a `car`.
- `diagonalmatrix` is a `matrix`.

The **Child Class** that inherits from the **Parent Class** will inherit all the code from the parent class.

In other words, **Child Class** reuses code from **Parent Class**.

Consider the following `student` class:

```
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 ...
};
```

Now, we want to create a `CSstudent` class.

We have **two goals**:

- We want to do so without duplicating the code.
- Our existing code written for `student` should work as is.

E.g., we want do

```
CSstudent s(1234,"lucy peng", 70);
lucy.set_grade(70);
```

We *almost* do it simply using (Public) Inheritance Syntax:

```
class child_name: public parent_name{};
```

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 *almost* whatever `student` class does.

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

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

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

```
class CSstudent: public student{
 int programming_grade;
 public:
 int get_programming_grade(){
  return programming_grade;
 }
 void set_programming_grade(int grd){
  if(grd <= 100 && grd > 0){
  programming_grade = grd;
  }
 }
};
```

Now, **in addition to** all fields and methods that are already in `student`, `CSstudent` has an extra field and two extra methods.

For example:

```
CSstudent lucy;
lucy.set_grade(70);

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

lucy.set_programming_grade(80);

printf("%d\n", lucy.get_programming_grade());
//prints out 80.
```

Moreover, all functions that take a `student` object as an input will now take `CSstudent` as input.

Since the C++ knows that `CSstudent` is a `student`.

Suppose we have a standalone function (not a method):

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

Now we can call `print_grade` using `lucy` as an input:

```
CSstudent lucy;
lucy.set_grade(70);
print_grade(lucy);
// OK, C++ knows lucy is a CSstudent, thus it
// deduces that lucy is a student and prints 70.
```

Constructors in Inheritance

Constructors are not inherited, even if the parent has one.

You need to rewrite constructors for child classes!

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

public:
student(int newid, const char* newname, int newgrade){
 ID = newid;
 name = newname;
 set_grade(newgrade);
}

// rest is omitted
};
```

If you create a `CSstudent` class inherits from `student`

```
class CSstudent: public student{
};
```

The following does not work!

```
CSstudent jack(1234, "lucy", 70); //compilation error!
```

The constructor is not inherited!

Compiler can not find a suitable constructor for the given list of input arguments!!

You can rewrite a constructor for your new class:

```
class CSstudent: public student{
 int programming_grade;

 public:
 CSstudent(int newid, const char* newname,
           int newgrade, int p_grade)
           :student(newid, newname, newgrade)
 // calling the constructor in the parent class
 {
  programming_grade = p_grade;
  // validity checking omitted
 }
};
```

Now you can write:

```
CSstudent jack(1234, "lucy", 70, 90); // OK!
```

Other Issues in Inheritance

This lecture only aims to give you a glimpse of OOP in C++.

There are many other aspects you need to master before writing a large scale OOP program.

If you are interested, read here for more information about inheritance.

Conclusion

Structures in C has some issues:

- Code cannot be easily reused.
- 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 data and functions.

C++ features:

- It is a superset of C.
- Class and Objects
- Fields and Methods
- Constructor/Destructor
- Inheritance

Homework 1

You will need the matrix class you wrote in the previous lab.

You can check your answer with the solution from last week, see `lab_matrix_sol.cpp`.

Homework 1

Start from the `lab_matrix_2_template.cpp` file.

1. Modify your **constructor** and add a **destructor** so that:

- Your `matrix` class manages its own memory.
- The constructor takes only **two input arguments**: `nrow` and `ncol`. It allocates heap memory for your matrix.
- The destructor releases the heap memory allocated for your matrix.

2. Write two public methods in the class:

- `int get_num_rows()` and `int get_num_cols()` returns number of rows and columns of the matrix.

Homework 2 (submit)

1. Images are essentially matrices stored in the memory, so you can say, an image "is a" matrix.
2. Create an `image` class that inherits from the `matrix` class.

Homework 3 (submit)

3. Write an constructor for the `image` class:

- `image(int height, int width, int *data)`
- where `width` and `height` are the width and height of the image, `data` array stores a row-major matrix.
- In the constructor, you need to initialize **all fields** in the image class appropriately.
- You need to copy the data from `data` to `elements`.

4. Write a method `show()` in the `image` class:

- It prints the image on the screen as in the solution to lab 7 (TB1).
- See also https://github.com/anewgithubname/MATH10017-2023/blob/main/lecs/lec8.pdf

Homework 3 (submit)

5. Write code in `main` and test your `image` class.
6. Does your program have memory leak issue?

- Hint, print out a message each time you allocate the memory.
- print out a message each time you free the memory.
- Do you see the same number of messages for allocating the memory and freeing the memory?

Homework 4 (Challenge)

1. Write a stand-alone function (not a method)

```
matrix mult_matrices(matrix A, matrix B)
```

that computes the element-wise product of A and B.

2. Try to use it with your latest version of the `matrix` class, which manages its memory. Code such as:

```
int main()
{
matrix A(3, 3);
matrix B(3, 3);
matrix C = mult_matrices(A, B)
}
```

should give an error such as:

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

What happened?

3. Make sure that you print out a message every time you call the constructor/destructor. If you do that, you should see:

```
allocating memory for the matrix...
allocating memory for the matrix...
allocating memory for the matrix...
freeing the matrix...
freeing the matrix...
freeing the matrix...
freeing the matrix...
free(): double free detected in tcache 2
Aborted (core dumped)
```

Three allocations but four freeing operations?????

```
int main()
{
matrix A(3, 3);
matrix B(3, 3);
matrix C = mult_matrices(A, B)
// Using A and B from now on might crash your program!!
}
```

In `mult_matrices(matrix A, matrix B)` matrices `A` and `B` are passed by value, that is they are copied.

Now there is a copy of `A` and `B` in `main` and another in `mult_matrices`.

But the two copies of `A` contain the same pointer `*elements`.

When we exit `mult_matrices`, that memory is freed.

Now the original `A` (that one in `main`) is invalid.

When we exit `main` the destructor of `A` is called again!!

One solution is based on the so-called **copy constructor**.

It creates an instance of a class based on another instance of the same class.

The copy constructor is a public methods that looks like this

```
matrix(const matrix &M){
// Your code
}
```

In `mult_matrices(matrix A, matrix B)` the matrices are passed by value (i.e. copied).

To do that the default copy constructor will be called.

The default copy constructor does not work well when your class contains pointers!

4. Add to you family a copy constructor

```
matrix(const matrix &M){
 // Initialise numrows and numcols using M
 // Allocate memory for elements
 // Copy elements from M to elements of current object
 // NOTE: constructor has access to private elements of M.
}
```

Don't worry about const and &, treat M as a regular variable.

5. Test whether `mult_matrices` now works.
6. Previously we have added the `void add(matrix B)` method to our class, test whether that works too.

You can read more here and here.