# 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("doing matrix stuff\n");
    return 0;
}
```

The 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.
  - 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, 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:

```cpp
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`.

```cpp
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.

# Homework 2 Image Class.

- Since images are essentially matrices when stored in computer, we can create a `Image` class using the `Matrix` class we have already made last week.
- Let us create a child class `Image` by inheriting the `Matrix` class.
  - an `Image` is a `Matrix` in the same sense that `CSstudent` is a `student`,
  - The `Image` class inheriting the `Matrix` class can access all fields and functions in `Matrix`.

# Homework 2 Image Class

- **Using the skeleton code provided**, create a new child class called `Image` which **inherits** `Matrix` class.
- Write public member functions `int height()` and `int width` for the `Image` class, they return the height and width of the image.
  - Hint: If the image is represented by a numerical matrix, the height of the image is the number of rows of the matrix and the width of the image is the number of columns in the matrix.

# Homework 2 Image Class

- Write a public member function `void visualize()` in this `Image` class, visualizing the image represented by printing the matrix.
- Suppose an `Image` object is also a matrix $A$,
  - If $A_{i,j} \leq 85$, print `' '`.
  - If $85 < A_{i,j} \leq 170$, print `I`.
  - If $170 < A_{i,j} \leq 255$, print `M`.

## Homework 2 Image Class

- Try to visualize the image stored in `image.matrix`.