C++

Introduction

Brief history of C++

- C++ was created by Bjarne Stroustrup at AT&T Bell labs in the 80s
  - A “first version” was an extension of C with classes inspired from Simula
  - C++ was designed in 1985
  - The version 2.0 in 1989
  - C++ was standardized in 1998
  - The standard for the next version of the language is currently in development

Brief history of C++ - 2

- The design of C++ was guided by the following principles:
  - The use of classes should not result in programs executing (significantly) more slowly than programs not using classes
  - C programs should compile and run as subsets of C++ programs
  - No run-time inefficiency should be added to the language
- In summary: allow for higher level constructs than languages like C without penalty in term of speed

A simplified history of languages

- Early programming languages like Fortran (old versions) or Basic were unstructured (“spaghetti code”)
  - Use of GOTO for loop and branching
  - Code difficult to debug, maintain and extend
- Next generation: structured programming.
  - Example of languages: Pascal, C, Algol
- Object Oriented Programming: Simplify the development of large, consistent code
  - Example of languages: Java, Smalltalk, Eiffel, C++

Example of spaghetti code

- hard to follow the flow
- difficult to extend, maintain or debug

```
      DO 110, I = 1, N
               IF (I .EQ. H) THEN
                  XP(I) = X(I) + RAND()
               ELSE
                  XP(I) = X(I)
               END IF

         IF((XP(I) .LT. LB(I)) .OR. (XP(I) .GT. UB(I))) THEN
                  XP(I) = LB(I) + (UB(I) - LB(I))*RAND()
                  IF(IPRINT .GE. 3) GOTO 200
               END IF
110   CONTINUE

C rest of the code …..

200   PRT3(MAX, N, XP, X, FP, F)
```

Example of structured programming

```
int int_pow(int x, unsigned int n) {
  int x2 = x;
  if (n==0) return 1;
  if (n==1) return x;

  while (n > 1) {
    x2 = x * x2;
    n--;
  }
  return x2;
}
```

```
int main (int argc, char** argv) {
  unsigned int n = 3;
  int x = 2;
  printf (“%d¥n”, int_pow(x,n));
  return 0;
}
```

- Loop constructs: for, while, do-while
- Branching: if / else, switch / case
- Program defined by a sequence of procedure and / or functions

Problems of structured programming

- Difficulty to protect data and maintain consistency of data
- Data consistency: every time the value of a member of a data structure is changed, validity of data needs to be checked

```
typedef struct {
  int speed;
  int weight;
  int fuel_level;
} Car;
```

Example:
- when accelerating, speed can not be greater than the maximum speed of the car
- when adding fuel, the fuel level can not be greater than the tank capacity
- adding fuel will increase the weight of the car

Data consistency

- Ensuring data consistency in a program is one of the challenges of building robust software
- It is even more difficult as the program is modified over time, new constraints are added, data members are added and other persons (than the developer) will use the code

Solution

- Possible solution: define a restricted set of functions that access the data members and are carefully written to avoid inconsistency
- Other developers can access data only through these functions

```
typedef struct {
  int speed;
  int weight;
  int fuel_level;
} Car;
```

```
void accelerate(Car my_car, int increase) {
  int new_speed = my_car.speed + increase;
  if (new_speed > MAX_SPEED){
    …..
  }
}
```

Discussion of the solution

- If we can enforce that only these functions are used to access data, then modifying the Car structure is easier
- We only need to modify the restricted number of functions that directly access the data and make sure that they do not violate the constraints
- The problem is that we can not ensure that the data members are only accessed by the restricted set of functions in a procedural language like for example C

Object-Oriented Programming

- Enforcing data access by a restricted set of functions is solved by object-oriented programming (OOP)
- In OOP the fundamental entity is a class
- A **class** is a user defined data type (a set of states and a set of operations which transition between those states)
- Classes expose an interface and can protect (hide) their data
- Classes are used through their interfaces

Object-Oriented programming

- Languages supporting OOP provide mechanisms to enforce:
  - Data abstraction
  - Encapsulation (hiding data and / or implementation)
  - Inheritance
- It is possible to simulate OOP in procedural languages like C but there are no mechanisms to enforce it

Example

```
class Car {
 public:
    void accelerate(int increase);

 private:
    int speed;
    int weight;
    int fuel_level;
};


void a_function ( ) {
 Car my_car;
 my_car.speed += 10; // impossible: caught by the compiler
 my_car.accelerate(10); // ok
}
```

Some terminology

- A **class** is a user defined data type (a set of states and a set of operations which transition between those states)
- An **object** is a region of storage with associated semantics
- **Instantiating** an object is the process of creating an object
- Each object of a class has its own data (with their own values)
- In OOP instead of calling procedure, we are **invoking methods** on an object