PART 2

Why Classes & Objects

C++ / Python Programming

Session #2

@Institut d'Astrophysique de Paris

OUTLINE

1. Structures
2. Classes
   a. Constructors
   b. Destructors
   c. Protected vs Private
   d. Data members
   e. Function member
   f. Dot (.) vs arrow (->) operator
3. Canonical Classes
   a. Copy Constructors
   b. Assignment Operator
4. Derived Classes
5. Polymorphism

PURPOSE

1. Why we need structures?
2. Why we should move to classes
3. Understanding classes
4. Inheritance and Polymorphism

Typing is good ....

- Suppose we have to implement a program for complex numbers!
- As we know, complex is composed of two floats, we use array:

```
float c [2];
c [0] = 0.5;
c [1] = 0.11; //This represents 0.5 + 0.11j
```

- Implement function to handle complex numbers:

```
void add_complex(float c1[2], float c2[2], float result[2]) {
  result [0] = c1 [0]  + c2 [0] ;
  result [1] = c1 [1] + c2 [0];
}
// Did you find bug?
```

Neither Function declaration nor Function body are easy to read

Typing is good ….

- Improvement can be:

```
#define REAL 0
#define IMAG 1
typedef float complex_t [2];
complex_t c;
c [REAL] = 0.5;
c [IMAG] = 0.11;
```

- Now function to handle complex numbers:

```
void add_complex(complex_t c1, complex_t c2, float result[2]) {
  result [REAL] = c1 [REAL]  + c2 [REAL] ;
  result [IMAG] = c1 [IMAG] + c2 [REAL];
}
// Now bug is visible
```

Function is more readable. isn't it?

Typing is good ....

- Now, We know complex number supports two forms: Cartesian and Polar.
  The improved program is:

```
#define REAL 0
#define IMAG 1
typedef float complex_t [2];
#define MOD 0
#define ARG 1
typedef float polar_t[2];
```

- ```
  complex_t c, d;
  read_complex(c);
  read_complex(d);
  polar_t x;
  convert_complex_to_polar (c, x)
  /*...Do something */
  complex_t y;
  add_complex(x, d, y);
  ```

types are not enough.
So, here comes Structure!!!

In add_complex, complex is used in polar form but it requires in cartesian form.
Here, Compiler wouldn't detect the type error!!!
Because, complex_t and polar_t both an array of 2 floats
  ```

Structure

- To store different types of variable under same name.

```
struct StructureName
{
 Type1 Variable_1;
 Type2 Variable_2;
};
```

- In order to access the members of the structure.

```
Variable_StructureName.Variable1;
```

- This Variable_StructureName is defined in function to access a structure.

```
StructureName Variable_StructureName;
```

```
1.  struct database {
2.    int id_number;
3.    int age;
4.    string Name;
5.  };

6.  int main() {
7.   database employee;
8.   employee.age = 22;
9.   employee.id_number = 1;
10.  employee.Name = “John”;
11. }
```

```
typedef struct {
  float real;
  float imag;
} complex_t;

typedef struct {
  float mod;
  float arg;
} polar_t;
```

Now, there are two types of structure and can be use as return type unlike arrays

```
complex_t add_complex(complex_t c1, complex_t c2) {
  complex_t result;
  result.re   al = c1.real + c2.real;
  result.imag = c1.imag + c2.imag;
  return result;
}
```

Functions and structure together

```
typedef struct {
  float real;
  float imag;
} complex_t;

complex_t add_complex(complex_t a,
complex_t b);
complex_t substract_complex(complex_t a,
complex_t b);
complex_t mul_complex(complex_t a,
complex_t b);
complex_t div_complex(complex_t a,
complex_t b);
complex_t conju_complex(complex_t c);
complex_t float_to_complex(float x);
```

```
// evaluates complex polynomial on real coefficients
complex_t func(float func[10], complex_t x)
{
  complex_t s = {0, 0};
  complex_t k = {1, 1};
  for (int i=0; i<10; i++) {
     s = complex_add(s, mul_complex (c,
float_to_complex(func[i])));
     k = complex_mul(k, x);
  }
 return s;
}
```

It's ugly and hard to read.
Also, it's boring to repeat complex again and again

Classes

- user defined type and can consists zero or more members (members can be data member, function member or type member)
  Data Member: representation of object
  Function Member: provides operation on the objects
  Type Member: any type associated with class
- specifies, how object are represented and operations that can be performed on objects
- Access specifier (control level of access to members): Public, Private and Protected
  - private: member can only be accessed by other members of class and friends of class (by default)
  - public: member can be accessed by any code
  - protected: relates to inheritance
- Structure is kinda class where all the members are public.

Classes

Typical Class declaration:

```
class MyClass // The class is named MyClass.
{
public:
// public members (i.e., the interface to users)
// usually functions and types (but not data)
private:
// private members (i.e., the implementation details only and accessible by members of class)
// usually functions, types, and data
};
```

Codes in block (a) & (b) and (c) & (d) shown below are same:

```
class SomeClass {
  //  ...
};
```

(a)

```
class SomeClass {
 Private:
  //  ...
};
```

(b)

```
struct  SomeClass {
  //  ...
};
```

(c)

```
class SomeClass {
 Public:
  //  ...
};
```

(d)

All in one entity

- class complex {
  complex add(complex c);
  complex sub (complex c);
  complex mul(complex c);
  complex div(complex c);
  complex conjug(void);
  float real, imag;
  };
- complex complex::add(complex c) {
  complex result;
  result.real = real + c.real;
  result.imag = imag + c.imag;
  return result;
  }

Here, add() operation takes complex c as a parameter but also an hidden parameter which is object itself on which this function will perform operation.

Constructors

- when new object created usually desirable to immediately initialize it to some known state
- prevents object from accidentally being used before it is initialized
- constructor has *same name as class*
- constructor has *no return type* (not even **void**)
- constructor *cannot be called directly* (although placement new provides mechanism for achieving similar effect, in rare cases when needed)
- constructor *can be overloaded*
- in certain circumstances, constructors may be automatically provided but sometimes, automatically provided constructors *will not* have correct behavior
- constructor that can be called with no arguments known as default constructor

Constructor Example

```
class Vector {                  // Two-dimensional vector class.
public:

 Vector() {                      // Default constructor
     x_ = 0.0; y_ = 0.0;
 }

 Vector(double x, double y) {   // Another constructor
       x_ = x; y_ = y;

 }
// ...

private:
  double x_;                    // The x component of the vector.
  double y_;                    // The y component of the vector.

};
Vector u;                       // calls Vector(); u set to (0,0)

Vector x();                     // declares function x that returns Vector

Vector v(1.0, 2.0);             // calls Vector(double, double)
```

Destructor

- This member function that is *automatically called* when object reaches end of lifetime in order to perform any necessary cleanup
- when object destroyed, must ensure that any resources associated with object (e.g., memory, files, devices, network connections, processes/threads) are released
- destructor for class T always has *name* T::~T
- destructor has *no return type* (not even **void**)
- destructor *cannot be overloaded*
- destructor always takes *no parameters*
- if *no destructor* is specified, destructor *automatically provided* that calls destructor for each data member of class type but sometimes, automatically provided destructor *will not* have correct behavior

Destructor Example

```
class MyClass {
public:
  // Constructor
  MyClass(int bufferSize) {              // allocate some memory for buffer
    bufferPtr = new char[bufferSize];
  }

  ~MyClass() {                                  // Destructor.
       // free memory previously allocated
       delete [] bufferPtr;
  }
// other constructor, assignment operator, ...
private:
char* bufferPtr;                  // pointer to start of buffer
 };
```

Protected VS Private

- The class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.
- The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Data Members

- class example:

```
class Vector_2 { // Two-dimensional vector class.
public:
  double x; // The x component of the vector.
  double y; // The y component of the vector.
};

void func() {
   Vector_2 v;
   v.x = 1.0; // Set data member x to 1.0
   v.y = 2.0; // Set data member y to 2.0
}
```

- above class has data members x and y

Static Data Members

- sometimes want to have object that is shared by all objects of class data member that is shared by all objects of class is called static data member
- to make data member static, declare using **static** qualifier
- static data member must (in most cases) be defined outside body of class
- example:

```
class Widget {
 public:
   Widget() {++count_;}
   Widget(const Widget&) {++count_;}
   Widget(Widget&&) {++count_;}
  ~Widget() {--count_;}
   // ...
 private:
    static int count_; // total number of Widget
                       // objects in existence
};
// Define (and initialize) count member.
int Widget::count_ = 0;
```

Static Data Members

- sometimes want to have member function that does not operate on objects of class
- member function of class that does not operate on object of class (i.e., has no **this** variable) called static member function
- to make member function static, declare using **static** qualifier
- Example:

```
class MyClass {
  public:
    // ...
    // convert degrees to radians
    static double degToRad(double deg) {  return (M_PI / 180.0) * deg; }
  private:
    // ...
};

void func() {
    double rad;
    rad = MyClass::degToRad(45.0);
    rad = x.degToRad(45.0);          // x is ignored
}
```

Function Members

- class example:

```
class Vector_2 { // Two-dimensional vector class.
public:
  double x; // The x component of the vector.
  double y; // The y component of the vector.
  void initialize(double x_, double y_);
};

void Vector_2::initialize(double x_, double y_) {
   x = x_;
   y = y_;
}

void func() {
   Vector_2 v; // Create Vector_2 called v
   v.initialize(1.0, 2.0); // Initialize v to (1.0, 2.0)
}
```

- above class has member function initialize
- to refer to member of class outside of class body must use *scope-resolution operator* (i.e., ::)

dot (.) and arrow (->) operator

- both used to reference individual members of classes, structures, and unions.
- dot operator is applied to the actual object.
- a.b is only used if b is a member of the object a. So for a.b, a will always be an actual object (or a reference to an object) of a class.
- Arrow operator is used with a pointer to an object.
- a->b is essentially a shorthand notation for (*a).b, i.e, if a is a pointer to an object, then a->b is accessing the property b of the object that points to.

```
#include<iostream>
class A {
 public:
   int b;
   A() { b = 5; }
};

int main() {
   A a = A();
   A* x = &a;
   std::cout << "a.b = " << a.b << "\n";
   std::cout << "x->b = " << x->b << "\n";
   return 0;
}
```

Canonical Class

A canonical class must obligatory own the following special member functions :

- A constructor by default
- A copying constructor
- A destructor
- Assignment operator

The standard C++ compiler generates automatically these methods if they are not provided. But if one constructor (resp. one destructor)is provided then no default constructor (resp. destructor) is automatically generated.

Copy Constructor

- used to create object by copying from already-existing object
- copy constructor for class T typically is of form T(**const** T&)
- if *no copy constructor* specified (and no move constructor or move assignment operator specified), copy constructor is *automatically provided* that copies each data member (using copy constructor for class and bitwise copy for built-in type)

Copy Constructor Example

```
class Vector {                  // Two-dimensional vector class.
public:
 Vector() {                      // Default constructor
     x_ = 0.0; y_ = 0.0;
 }

 Vector(const Vector& v) {      // Copy constructor.
      x_ = v.x_;
      y_ = v.y_;
 }
 Vector(double x, double y) {      // Another constructor
       x_ = x; y_ = y;
 }
// ...
private:
  double x_;                        // The x component of the vector.
  double y_;                        // The y component of the vector.
};

Vector v(1.0, 2.0);             // calls Vector(double, double)
Vector w(v);                    // calls Vector(const Vector&)
Vector z = v;                   // calls Vector(const Vector&)
```

Copy Assignment operator

- for class T, T::**operator**= having exactly one parameter that is lvalue reference to T known as copy assignment operator
- used to assign, to already-existing object, value of another object by *copying*
- if no copy assignment operator specified, copy assignment operator *automatically provided* that copy assigns to each data member (using data member's copy assignment operator for class and bitwise copy for built-in type)
- copy assignment operator for class T typically is of form T& **operator**=(**const** T&) (returning reference to ***this**)

Copy Assignment operator Example

```
class Vector {                  // Two-dimensional vector class.
public:
 Vector() {                      // Default constructor
     x_ = 0.0; y_ = 0.0;
 }

 Vector(const Vector& v) {      // Copy constructor.
      x_ = v.x_;
      y_ = v.y_;
 }

Vector& operator=(const Vector& v) { // Copy assign
   if (this != &v) {
      x_ = v.x_;
      y_ = v.y_;
    }
   return *this;
}
 Vector(double x, double y) {      // Another constructor
       x_ = x; y_ = y;
 }
// ...
private:
  double x_;                       // The x component of the vector.
  double y_;                       // The y component of the vector.
};
```

```
Vector v(1.0, 2.0); // calls Vector(double, double)
Vector w(v); // calls Vector(const Vector&)
Vector z = v; // calls Vector(const Vector&)
Vector u(1.5, 2.5);
u = v; // calls copy assign
```

Exercice : class (1/3)

Download Point class and rectangle.h from the Exercises/download

Write a rectangle class such as rectangle.h. A rectangle will be define by 2 points.

Exercises (2 / 3)

1. Transform the previous Rectangle class in a canonical one
2. Test the obtained code with the following main :

Exercises (3/3)

```
1.  void function1(Rectangle r){
2.      r.affiche_info();
3.  }
4.  void function2(Rectangle& r){
5.      r.affiche_info();
6.  }
7.  int main(){
8.      Point p1(1,2);
9.      Point p2(2,3);
10.     Rectangle r1;
11.     rectangle r2(p1, p2);
12.     Rectangle r3 = r2;
13.     r1= r2;
14.     function1(r2);
15.     function2(r2);
16.     return 0;
17.}
```

- Which special method of the rectangle class is called line 10, 11, 12,13,14, 15 and 16 ?

line 10 : constructor by default
line 11 : exhaustive constructor
line 12 : copying constructor
line 13 : affectation operator
line 14 : copying constructor
line 15 : Nothing
line 8 : destructor

Derived Classes

- create new class from existing class by adding new members or replacing (i.e., hiding/overriding) existing members (feature known as *inheritance)*
- new class called derived class and original class called base class
- derived class said to inherit from base class
- can add new members (not in base class) to derived class
- syntax for specifying derived class:
  class *<derived class>* : *<base class specifiers>*

Person Class

```
#include <string>
class Person {
public:
  Person(const std::string& family_name, const std::string&
given_name) : family_name_(family_name),
given_name_(given_name) {}

  std::string family_name() const { return family_name_;}

  std::string given_name() const {return given_name_;}

  std::string full_name() const { return family_name_ + ", " +
given_name_;}
// ...
protected:
  std::string family_name_;
  std::string given_name_;
};
```

Student Class without Inheritance

```
class Student {

public:
  Student(const std::string& family_name, const
std::string& given_name) :
family_name_(family_name),
given_name_(given_name) {}
  std::string family_name() const {return
family_name_;}
  std::string given_name() const {return given_name_;}
  std::string full_name() const {return family_name_ +
", " + given_name_;}
  // NEW
  std::string student_id() {return student_id_;}

private:
  std::string family_name_;
  std::string given_name_;
  std::string student_id_;        // NEW
};
```

Student Class with Inheritance

```
class Student : public Person {
public:

  student_id_(student_id) {}


  std::string student_id() {return student_id_;}

  Student(const std::string& family_name, const std::string&
given_name, const std::string& student_id) :
Person(family_name, given_name),

private:
  std::string student_id_;

};
```

Types of Inheritance

```
class A  {
public:
    int x;
protected:
    int y;
private:
    int z;
};
class B : public A  {
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```
class C : protected A  {
    // x is protected
    // y is protected
    // z is not accessible from C
};
class D : private A  {  //
'private' is default for classes
    // x is private
    // y is private
    // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Polymorphism

- polymorphism is characteristic of being able to assign different meaning to something in different contexts

```
class ABC {
  public:
   // function with int parameter
   void func(int x) {
      cout << "value of x is " << x << endl;
   }
   // function with double parameter
   void func(double x) {
      cout << "value of x is " << x << endl;
   }
   // function with 2 int parameters
   void func(int x, int y) {
      cout << "value of x and y is " << x << ", " << y << endl;
   }
};
```

Conclusion

- You get the basis of classes and objects.
- Polymorphism is a main feature that make objects so powerful.
- Polymorphism + Inheritance is very powerful