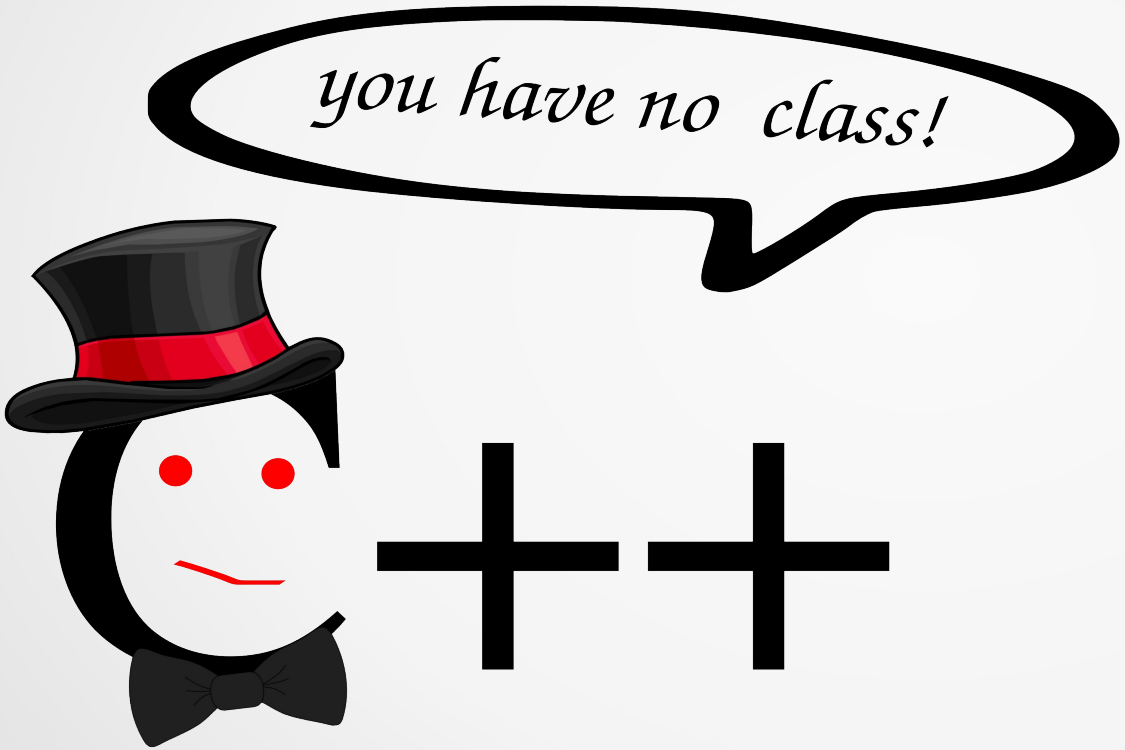
C++ Classes (Part I)



Krishna Kumar

Class Basics

- A class is a **user-defined type**.
- A class consists of a set of members. The most common kinds of members are **data members** and **member functions.**
- Member functions can define the meaning of initialization (creation), copy, move, and cleanup (destruction).
- Members are accessed using **. (dot) for objects and –> (arrow) for pointers.**
- Operators, such as +, !, and [], can be defined for a class.
- A class is a **namespace containing its members.**
- The **public members provide the class's interface** and the **private members provide implementation details**

Class basics

```
class X {
private:                        // the representation (implementation) is private
        int m;
public:                         // the user interface is public
        X(int i =0) :m{i} { }   // a constructor (initialize the data member m)

        int mf(int i)           // a member function
        {
                int old = m;
                m = i;          // set a new value
                return old;     // return the old value
        }
};

X var {7}; // a variable of type X, initialized to 7

int user(X var, X* ptr)
{
        int x = var.mf(7);          // access using . (dot)
        int y = ptr->mf(9);         // access using -> (arrow)
        int z = var.m;              // error: cannot access private member
}
```

Initialization function ()

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int) {
                    width = x; height =y; } ;
    int area () {return width*height;}
};
int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() ;
  cout << "rectb area: " << rectb.area();
}
```

- Output
  - rect area: 12
  - rectb area: 30
- What happens if the programmer forgets to call set_values() before calling area()?
  - An undetermined result

Constructor

```
class Date {
        int d, m, y;
public:
        // ...

        Date(int, int, int);        // day, month, year
        Date(int, int);             // day, month, today's year
        Date(int);                  // day, today's month and year
        Date();                     // default Date: today
        Date(const char*);          // date in string representation
};
```

```
Date today {4};                     // 4, today.m, today.y
Date july4 {"July 4, 1983"};
Date guy {5,11};                    // 5, November, today.y
Date now;                           // default initialized as today
Date start {};                      // default initialized as today
```

- declare a function with the explicit purpose of initializing objects.
- Such a function constructs values of a given type, it is called a constructor.
- A constructor is recognized by having the same name as the class itself.
- Use {} to represent intialisation over ().
- By guaranteeing proper initialization of objects, the constructors greatly simplify the implementation of member functions

Explicit vs Implicit Constructor

- By default, a constructor invoked by a single argument acts as an implicit conversion from its argument type to its type.
- complex<double> d {1}; // d == {1.0, 0}
- Such implicit conversions can be useful.
- However in many cases, such conversions can be significan source of confusion and errors.
- An initialization with an = is considered *copy initialization.* Initializer is placed into the initialized object.
- Leaving out the = makes the initialization explicit. It is known as *direct initialization.*

Member initialization in constructors

```
class Rectangle {
    int width, height;
  public:
    Rectangle(int,int);
    int area() {
        return width*height;
    }
};
```

```
Rectangle::Rectangle (int x,
int y) { width=x; height=y; }
```

- member initialization as:

```
Rectangle::Rectangle (int x,
int y) : width(x) { height=y; }
```

- Or even:

```
Rectangle::Rectangle (int x,
int y) : width(x), height(y) { }
```

Delegating constructors

```
class Foo
{
Public:
  Foo() { // code to do A }
  Foo(int nvalue) {
    // code to do A
    // code to do B
}
};
```

- In C++, it would be useful for one constructor to call another constructor in the same class.
- This is disallowed in C++0X
- This results in dublicate code.
- Here the “code to do A” is defined twice.

Delegating constructor (cont...)

- Use init() non-ctor function

```
class Foo {
Public:
  Foo() { initA(); }
  Foo(int nvalue) {
    initA();
    // code to do B
}
  void initA() { // code to do A}
};
```

- It is quite readable, but:
  - Adds a new function and several function calls.
  - initA() is not a constructor, it can be called during the normal program flow, where member variables and dynamic memory are set and allocated

Delegating constructors (cont...)

- C++11

```
class Foo {
Public:
  Foo() { // code to do A }
  Foo(int nvalue) : Foo ()
// use Foo() default ctor to do A
{
    // code to do B
}
};
```

- It is much cleaner!
- Use the initialization syntax when delegating ctor. Compilers which do not support delegating ctor will flag this as error.
- If you call one ctor from the body of another ctor, the compiler will not complain and your code may misbehave.

References

-
- Exceptional C++ - Herb Sutter