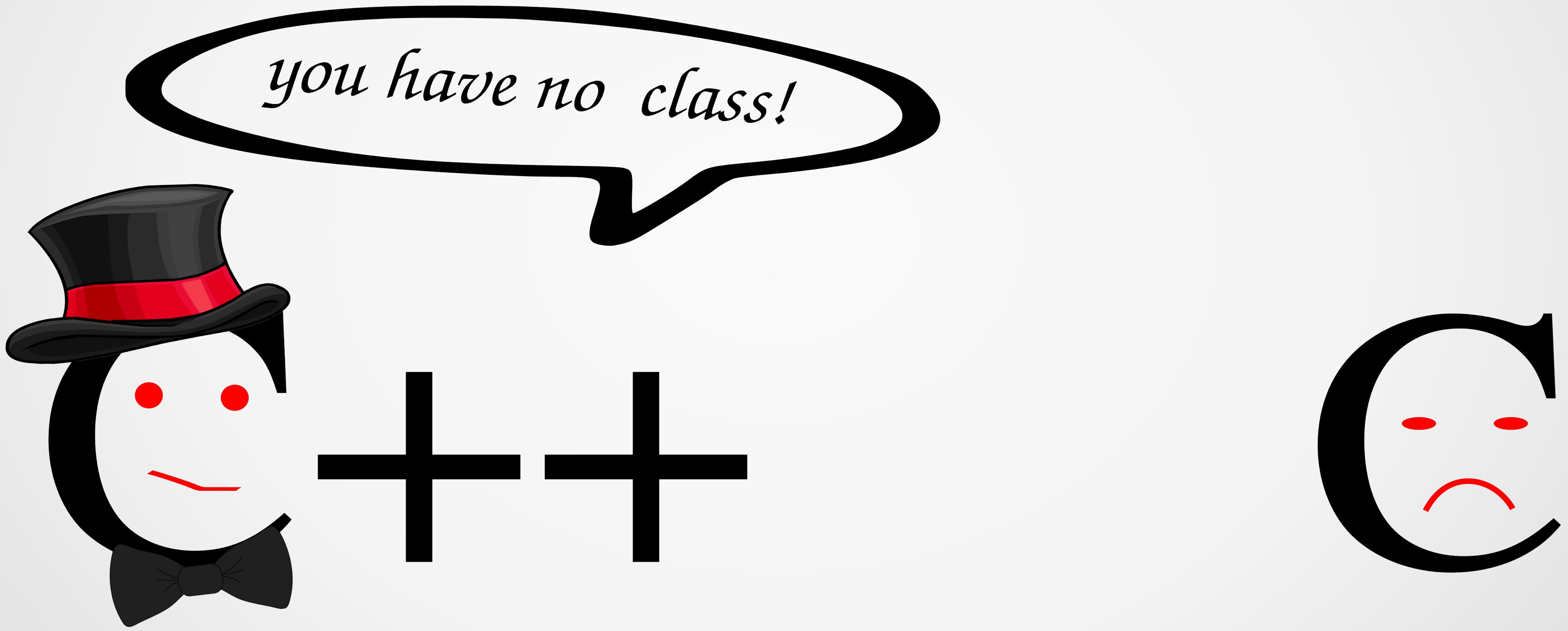
C++ Classes (Part I)

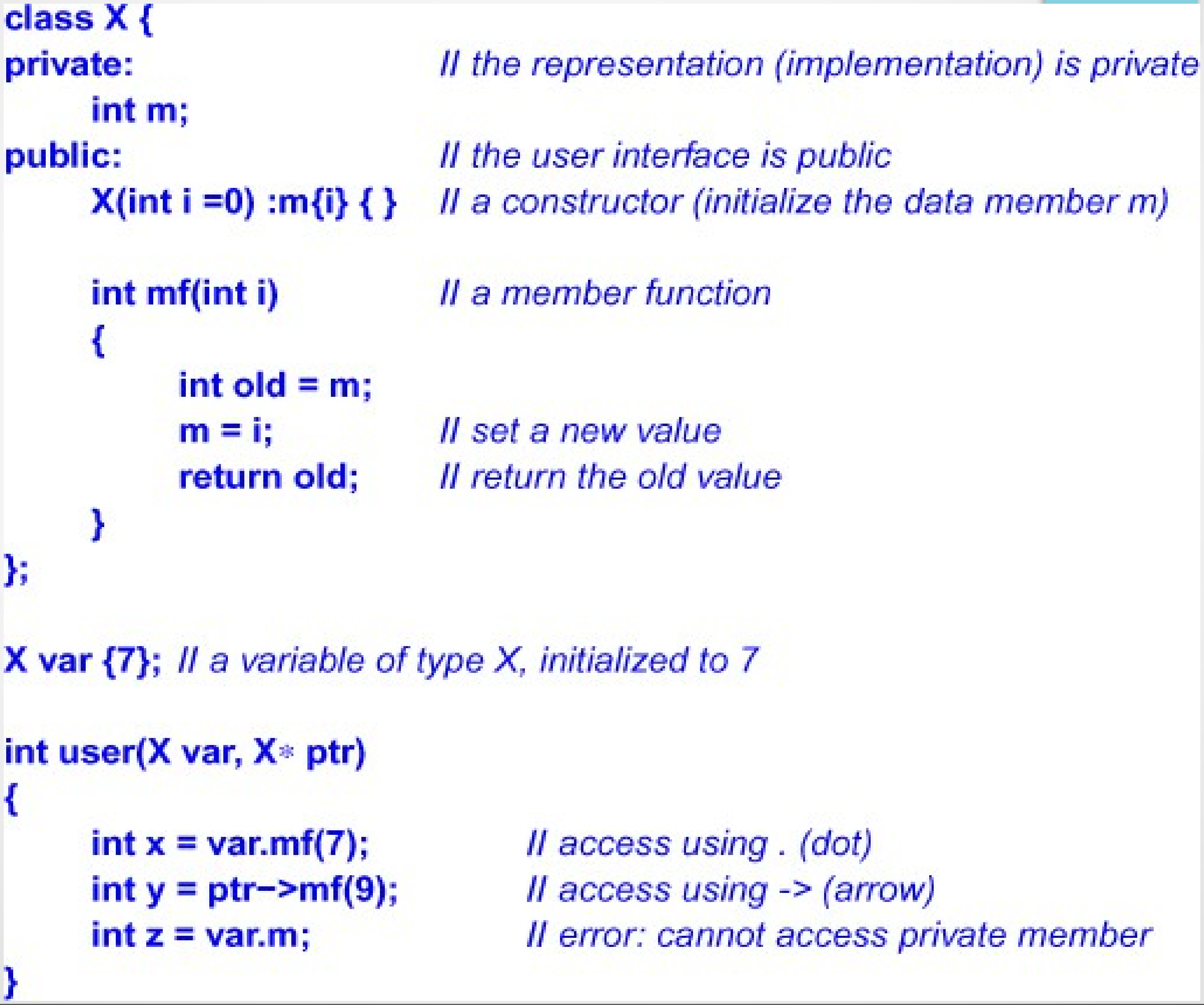


Krishna Kumar

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
}
```

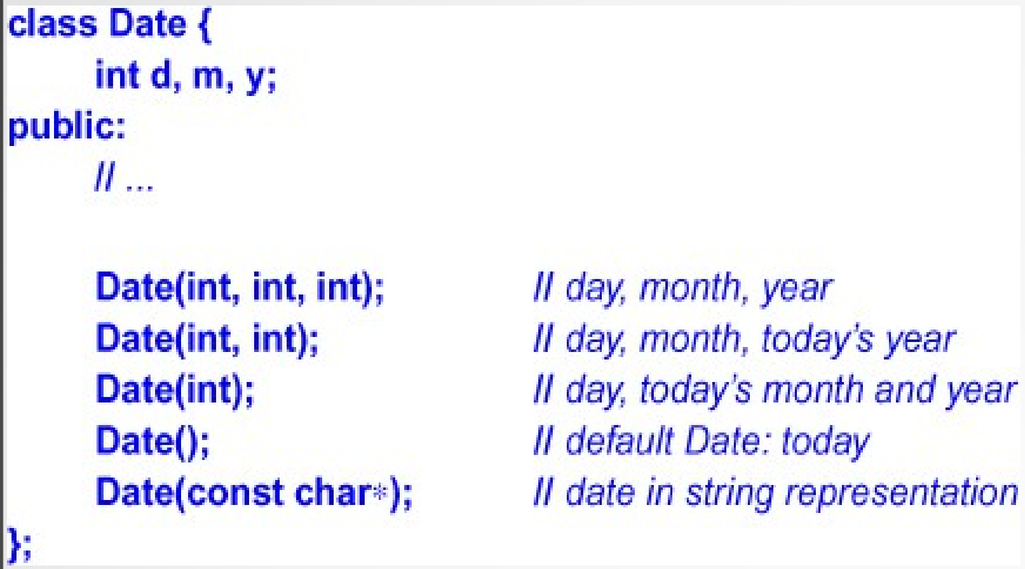
Initialization function ()

```
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

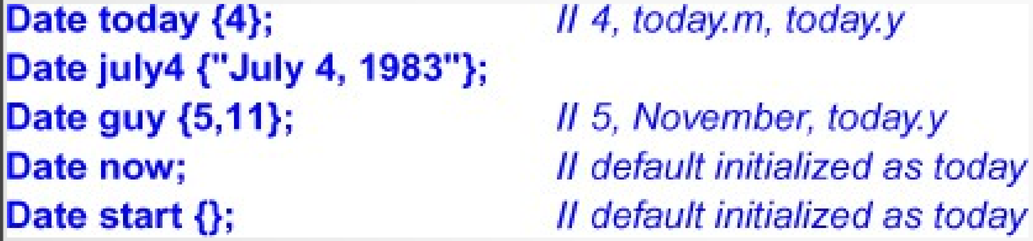
- What happens if the programmer forgets to call set_values() before calling area()?
  - An undetermined result

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.

Class Mechanics (Error 1)

```
Complex( double real, double imaginary = 0 )
: _real(real), _imaginary(imaginary)
{
}
```

Guideline:

- *Watch out for hidden temporaries created by implicit conversions. One good way to avoid this is to make constructors explicit when possible, and avoid writing conversion operators.*

- Constructor allows implicit conversion!
- Because the second parameter has a default value, this function can be used as a single parameter constructor and, hence, as an implicit conversion from *double* to *complex*
-

Error 2: operator+ is probably slightly inefficient.

```
void operator+ ( ( Complex other )
{
_real = _real + other._real;
_imaginary = _imaginary +
other._imaginary;
}
```

Guide:

Prefer passing objects by const& instead of passing by value

- Also, in both cases, "a=a+b" should be rewritten "a+=b". This improve efficiency when adding class types.
- For efficiency, the parameter should be passed by reference to const
- Prefer writing "a op= b;" instead of "a = a op b;" (where op stands for any operator). It's clearer, and it's often more efficient.

Error 3: operator+ should not be a member function.

```
void operator+ Complex other
{
_real = _real + other._real;
_imaginary = _imaginary +
other._imaginary;
```

Guide:

If you supply a standalone version of an operator (for example, operator+) always supply an assignment version of the same operator (for example, operator+=) and prefer implementing the former in terms of the latter

If operator+ is made a member function, as it is here, then it won't work as naturally as your users may expect when you do decide to allow implicit conversions from other types. Specifically, when adding Complex objects to numeric values, you can write "a = b + 1.0" but not

"a = 1.0 + b" because a member operator+ requires a Complex (and not a const double) as its left-hand argument.

It may make sense to provide the overloaded functions operator+(const Complex&, double) and operator+ (double, const Complex&) too.

Error 4:

```
void operator+ ( Complex other )
{
_real = _real + other._real;
_imaginary= _imaginary + other._imaginary;
}
```

operator+ should not modify this object's value, and it should return a temporary object containing the sum.

Note that the return type for the temporary should be "const Complex" (not just "Complex") in order to prevent usage like "a+b=c".

Error 5: operator<< should not be a member function

```
void operator<< ( ostream os )
{
os << "(" << _real << "," <<
_imaginary << ")";
}
```

Also, the parameters should be "(ostream&, const Complex&)". Note that the nonmember operator<< should normally be implemented in terms of a(n often virtual) member function that does the work, usually named something like Print().

Error 6:

Further, operator<< should have a return type of "ostream&" and should return a reference to the stream in order to permit chaining. That way, users can use y operator<< naturally in code like "std::cout << a << b;".

Error 7: The preincrement operator's return type is incorrect.

```
Complex operator++()
{
++_real;
return *this;
}
```

Preincrement should return a reference to non-const—in this case, Complex&. This lets client code operate more intuitively and avoids needless inefficiency.

Error 8: The postincrement operator's return type is incorrect.

```
Complex operator++( int )
{
Complex temp = *this;
++_real;
return temp;
}
```

Postincrement should return a const value—in this case, const Complex. By not allowing changes to the returned object, we prevent questionable code like "a++++", which doesn't do whata naïve user might think it does.

References

-
- Exceptional C++ - Herb Sutter