Next time: Read TCS, ch. 16.

Extensible Languages:

Extensible language have a core or base language and an extension mechanism, which allows you to add to the base language.

What computer scientists discovered was that it's too difficult to have a language that's good for all applications. You get "Swiss Army Knife" languages. Not a good idea. It's also not a good idea to have a separate language for every application area, because then there are too many languages to learn, and what do you do about applications that combine several application areas?

One solution is an extensible language, which has a core language that is more or less neutral as far as application areas go, but can be extended in various to make it more suitable for different application. The extension usually allows you to extend the syntax and types of the language (i.e., it goes beyond just function libraries).

What about C++. It's an extensible language. The core language is C plus a little more. The extension mechanism includes classes, user-defined types (strucs, enums), and overloading. So in C++ the class is the principal extension mechanism.

Overloading Operators:

The built-in C++ operators (+, -, *, /, %, ==, !=, >, >=, ++, etc.) have multiple meanings, depending the types of their arguments. For example, "+" can do either integer or floating point addition.

You can also overload additional meanings on these operators. For example, the <string> library overloads string concatenation on "+".

There are two ways to overload operators, as nonmember (or standalone) functions or as member functions. Suppose I want to overload "+" to work on complex numbers. So I want to be able to write "Z1 + Z2" where Z1 and Z2 are complex numbers (and get a complex number result).

I can define it like a standalone function:

```
Complex operator + (const Complex& a, const Complex& b)
{ return Complex (a.real + b.real, a.imag+b.imag); }
```

So in this case, "Z1 + Z2" is interpreted like a function call, "+ (Z1, Z2)". The problem, is since this is a nonmember function, it does not have access to the private variables of functions of Complex.

Alternately, you could make "+" a member function of Complex:

```
class Complex {
….
public:
Complex operator + (const Complex& b) const {
  return Complex (real + b.real, imag + b.imag);
}

}; \\ Complex class
```

In this case, the operation "Z1 + Z2" is interpreted as a call on a member function

of Z1: "Z1.+(Z2)"

The advantage is that "+" is part of the definition of Complex, and so it has direct access to the private member variables.

Overloading Insert (<<):

When you write "cout << N" that calls the "<<" (insert) operator with two arguments, an output stream (cout in this case), and something to insert into that output stream (N in this case).

But you know that you chain inserts together:

cout << "Ans = " << N << endl;

This works because << associates to the left and return its left argument as its values. So the above means:

(((cout << "Ans = ") << N) << endl);

Working from inside out, (cout << "Ans = ") inserts the string "Ans = " into the stream cout, and returns cout as its value. So the value of the inner parens is cout, and the effect is:

((cout << N) << endl);

So this inserts N into cout and returns cout, to get

(cout << endl);

How do you want to print out Complex numbers? What do you do if the imaginary part is negative?
What do you do if the imaginary part is zero?
What do you do if the real part is zero?
What do you do if they are both zero?
It's important to make sure you address all the possibilities!

The same applies to the extract operator (>>). "cin >> N" extracts from an input stream cin into N and returns the stream. So ideally whenever you define a new data type, you should define insert and extract operators to go with it. (Defining extractions is more complicated because you have to parse and error-check the input.)