Need of operator overloading

Operator overloading **facilitates the specification of user-defined implementation for operations wherein one or both operands are of user-defined class or structure type**. This helps user-defined types to behave much like the fundamental primitive data types.

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

Example:

```
       int a;
      float b,sum;
      sum=a+b;
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0) {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

Output:

```
12 + i9
```

Operators that can be overloaded

We can overload
Unary operators
Binary operators
Special operators ([ ], () etc)

But, among them, there are some operators that cannot be overloaded. They are

Scope resolution operator	: :
Member selection operator	
Member selection through	*
Pointer to member variable	
Conditional operator	? :
Sizeof operator	sizeof()

Operators that can be overloaded
Binary Arithmetic -> +, -, *, /, %
Unary Arithmetic -> +, -, ++, —
Assignment -> =, +=,*=, /=,-=, %=
Bit- wise -> & , | , << , >> , ~ , ^
De-referencing -> (->)
Dynamic memory allocation and De-allocation -> New, delete
Subscript -> [ ]
Function call -> ()
Logical -> &, | |, !
Relational -> >, < , = =, <=, >=

Why can't the above-stated operators be overloaded?
1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

```
#include <iostream>
using namespace std;

class ComplexNumber{
  private:
  int real;
  int imaginary;
  public:
  ComplexNumber(int real, int imaginary){
    this->real = real;
    this->imaginary = imaginary;
  }
  void print(){
    cout<<real<<" + i"<<imaginary;
  }
  ComplexNumber operator+ (ComplexNumber c2){
    ComplexNumber c3(0,0);
    c3.real = this->real+c2.real;
    c3.imaginary = this->imaginary + c2.imaginary;
    return c3;
  }
};
int main() {
    ComplexNumber c1(3,5);
    ComplexNumber c2(2,4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}
```

Output:

```
5 + i9
```