SSE2034: System Software Experiment 3
Spring 2016

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(const Rectangle &r);
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

- Default constructor
- Copy constructor
- Constructor with parameters

Constructor overloading

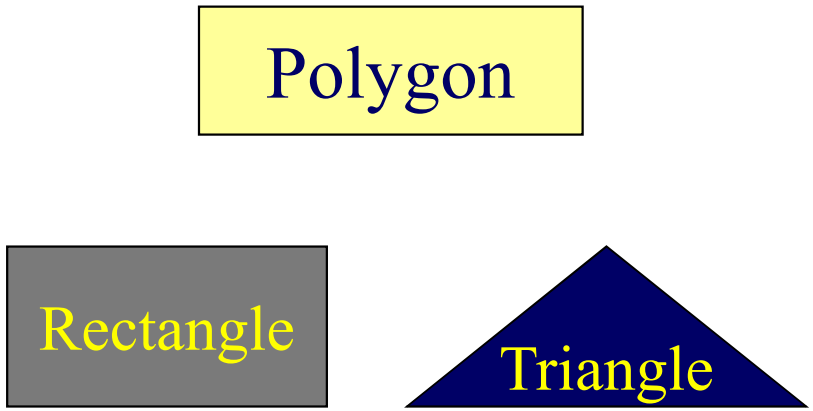
Overloading

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

void printstr(string str) {
       cout << str << endl;
}

void printstr(char c) {
       cout << c << endl;
}
```

Function overloading

Multiple definitions for the same function name

Inheritance Concept

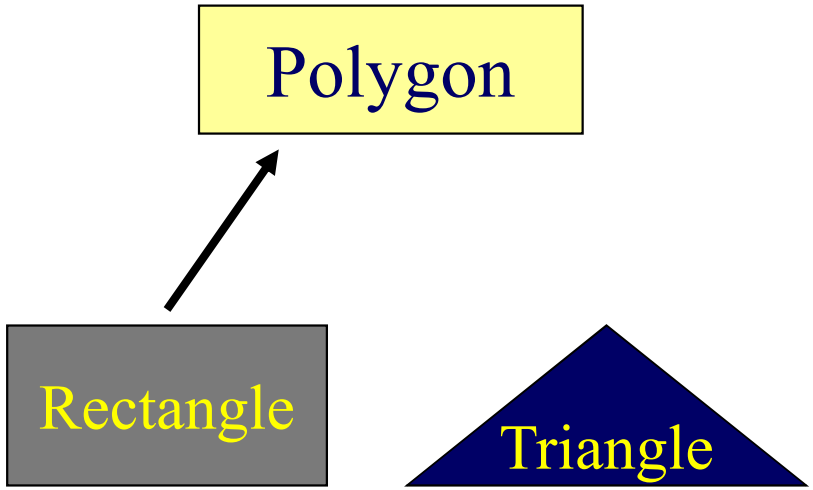



```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

```
class Polygon{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

```
class Triangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Inheritance Concept




```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
       float area();
};
```

```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Why Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
  - specialization
  - augmentation

  of existing types

Define a Class Hierarchy

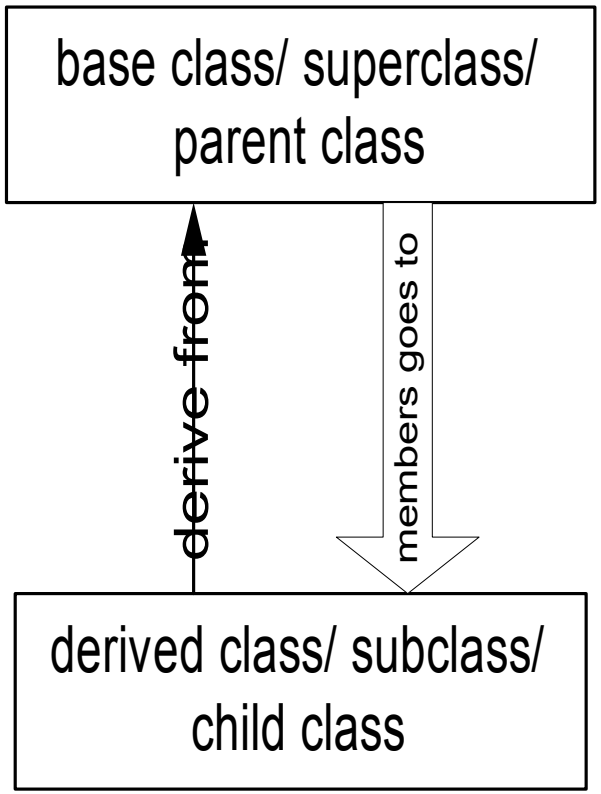
- **Syntax:**

  class ***DerivedClassName*** **: access-level** ***BaseClassName***

  where

  - access-level specifies the type of derivation
    - private by default, or
    - public
- **Any class can serve as a base class**
  - Thus a derived class can also be a base class

Access Control Over the Members



- **Two levels of access control over class members**
  - class definition
  - inheritance type

```
class Point{
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point{
    … …
};
```

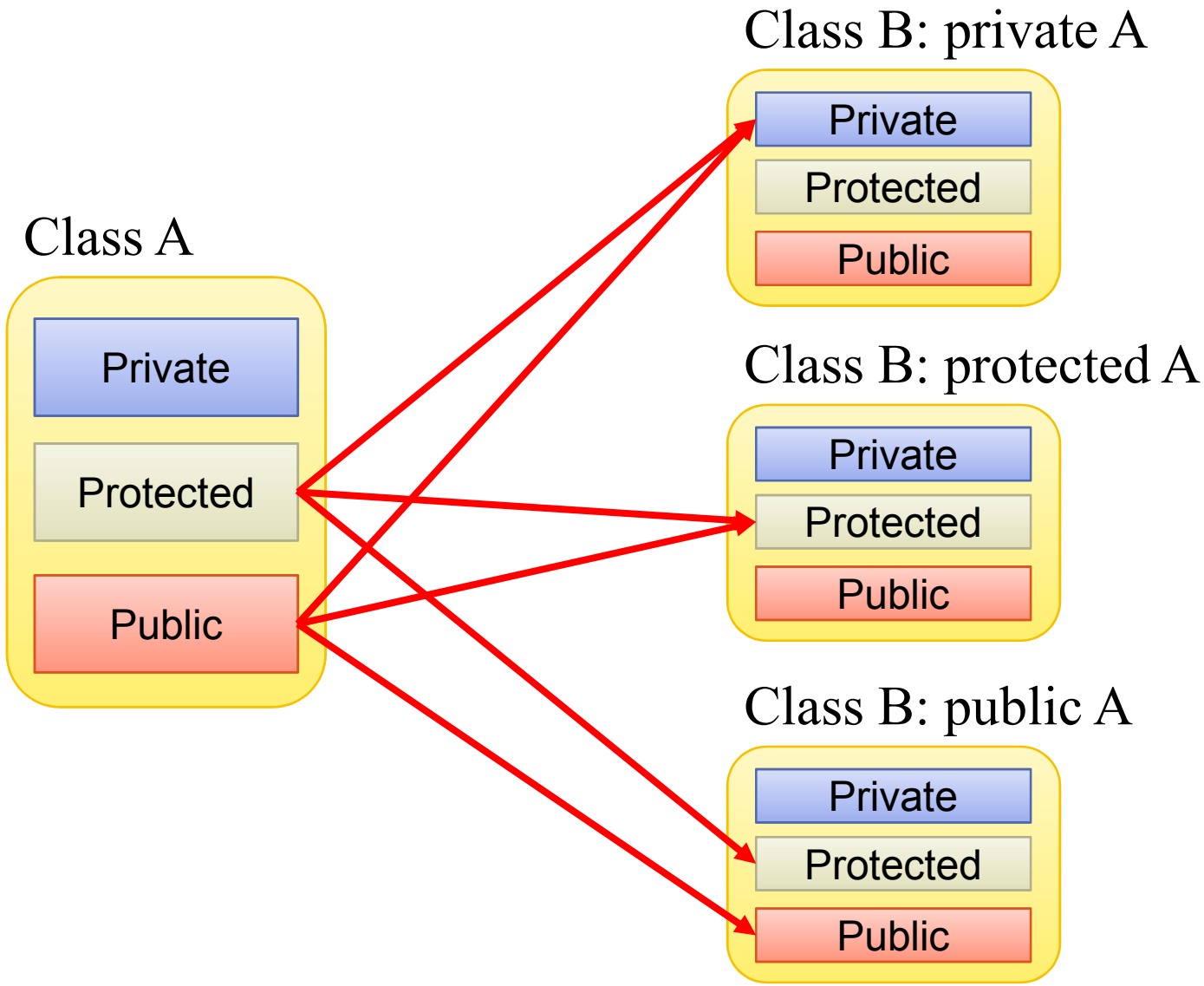
Access Rights of Derived Classes

Type of Inheritance

Access Control for Members	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

- **The type of inheritance defines the access level for the members of derived class that are inherited from the base class**

Class Derivation



Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
    A ( )
      {cout<< "A:default"<<endl;}
    A (int a)
      {cout<<"A:parameter"<<endl;}
};
```

```
class B : public A
{
  public:
    B (int a)
        {cout<<"B"<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify an constructor of the base class other than the default constructor

```
DerivedClassCon ( derivedClass args ) : BaseClassCon ( baseClass args )
        {  DerivedClass constructor body   }
```

```
class A {
  public:
    A ( )
      {cout<< "A:default"<<endl;}
    A (int a)
      {cout<<"A:parameter"<<endl;}
};
```

```
class C : public A {
  public:
    C (int a) : A(a)
        {cout<<"C"<<endl;}
};
```

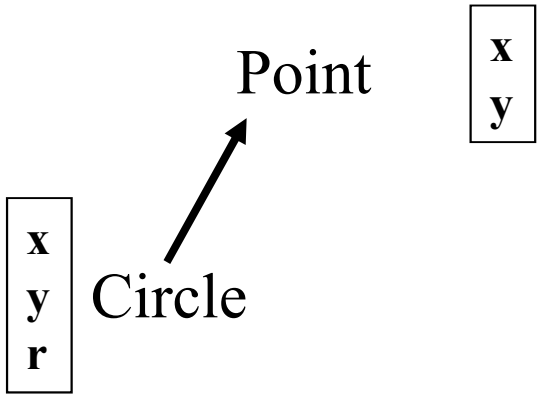
```
C test(1);
```

output:

```
A:parameter
C
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class




```
class Circle : public Point{
    private:
        double r;
    public:
        void set_r(double c);
};
```

```
class Point{
    protected:
        int x, y;
    public:
        void set(int a, int b);
};
```

```
class Circle{
    protected:
        int x, y;
    private:
        double r;
    public:
        void set(int a, int b);
        void set_r(double c);
};
```

Overriding

- **A derived class can override methods defined in its parent class. With overriding,**
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

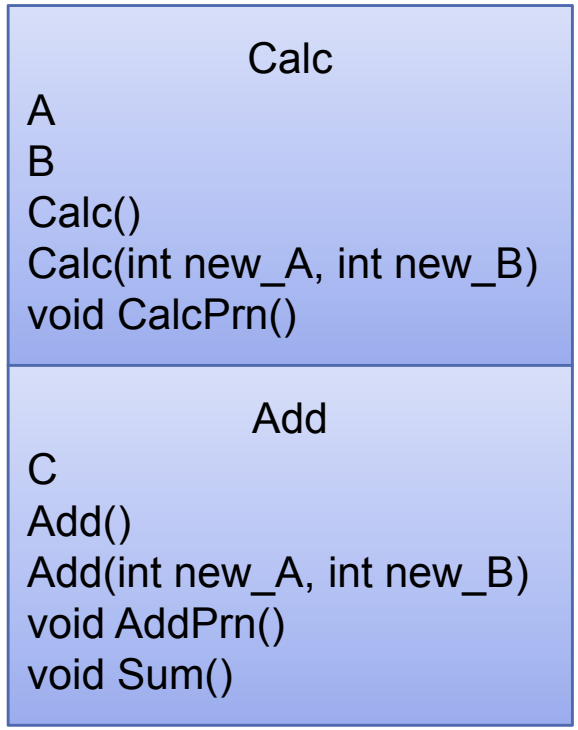
```
class A {
  protected:
    int x, y;
  public:
    void print ()
          {cout<<"From A"<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<"From B"<<endl;}
};
```

Upcasting

- **Converting a derived-class reference or pointer to a base-class**

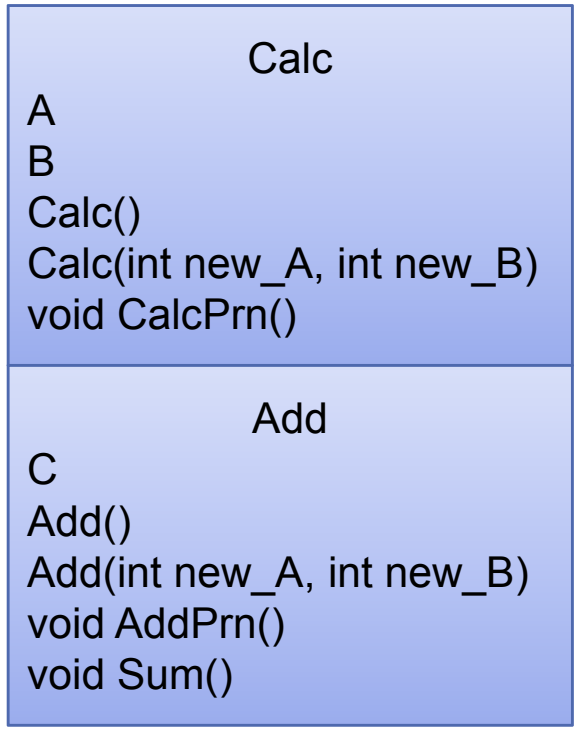
```
#include <iostream>
Using namespace std;

int main(){
        Add Addobj(3, 5)
        Calc *CalcPtr;

        CalcPtr = $Addobj;
        CalcPtr->CalcPrn();
}
```



Downcasting

- **Converting a base-class pointer to a derived-class pointer**

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

int main(){
        Calc *CalcPtr;
        CalcPtr = new Add(3,5);

        Add *AddPtr;
        AddPtr=(Add *)CalcPtr;
}
```



Operator overloading

- **Programmer can use some operator symbols to define special member functions of a class**
- **Provides convenient notations for object behaviors**

Why Operator Overloading

```
int i, j, k;          // integers
float m, n, p;        // floats

k = i + j;
    // integer addition and assignment
p = m + n;
    // floating addition and assignment
```

The compiler overloads the + operator for built-in integer and float types by default, producing integer addition with i+j, and floating addition with m+n.

We can make object operation look like individual int variable operation, using operator functions

```
Complex a,b,c;
c = a + b;
```

Operator Overloading Syntax

- **Syntax is:**

operator@*(argument-list)*

--- operator is a function

--- @ is one of C++ operator symbols (+, -, =, etc..)

Examples:

operator+

operator-

operator*

operator/

Example of Operator Overloading

```
class CStr
{
    char *pData;
    int nLength;
  public:
    // …
    void cat(char *s);
    // …
     CStr operator+(CStr str1, CStr str2);
     CStr operator+(CStr str, char *s);
     CStr operator+(char *s, CStr str);

     //accessors
    char* get_Data();
    int get_Len();

};
```

```
void CStr::cat(char *s)
{
   int n;
   char *pTemp;
   n=strlen(s);
   if (n==0) return;

   pTemp=new char[n+nLength+1];
   if (pData)
      strcpy(pTemp,pData);

   strcat(pTemp,s);
   pData=pTemp;
   nLength+=n;
}
```

The Addition (+) Operator

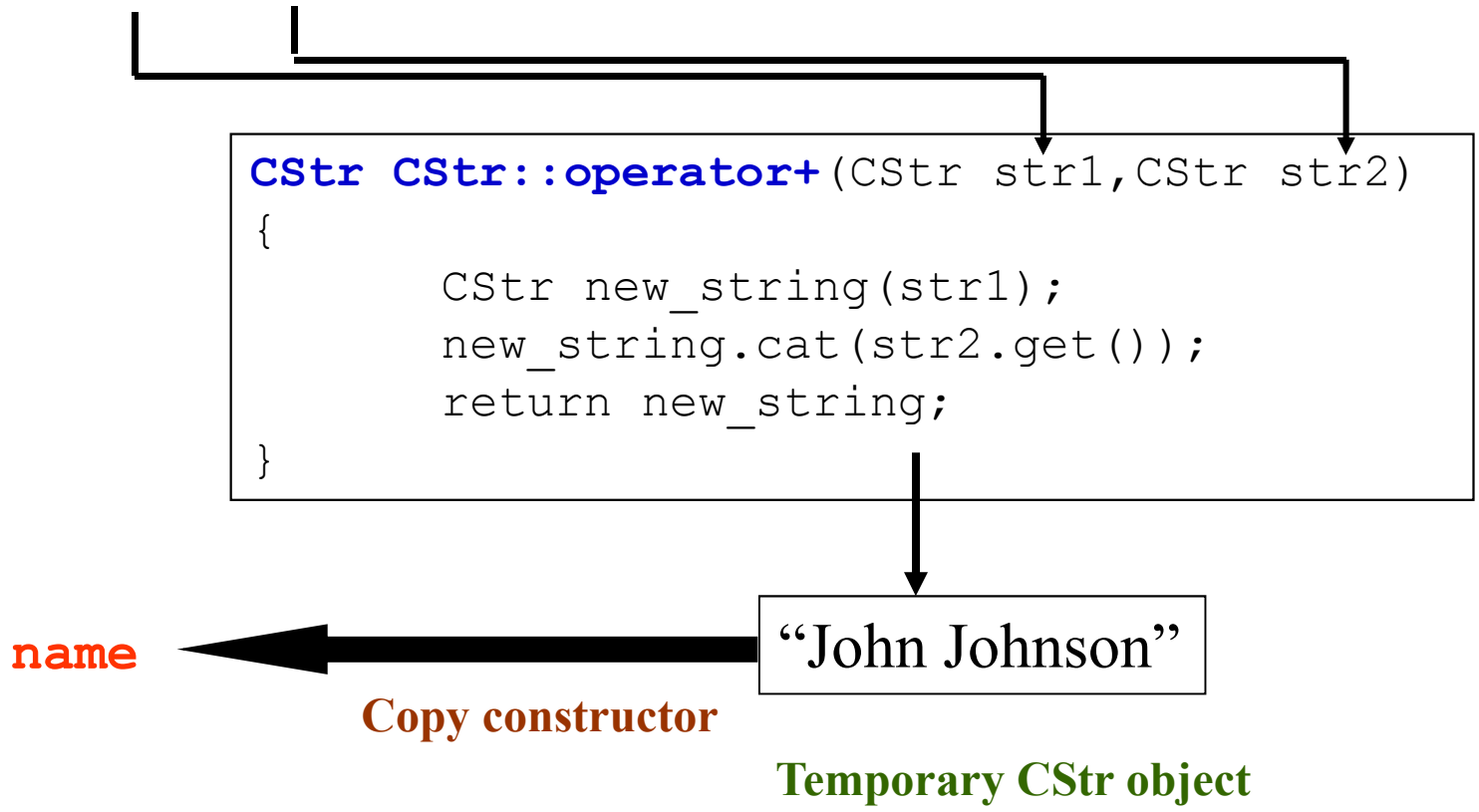
```
CStr CStr::operator+(CStr str1, CStr str2)
{
  CStr new_string(str1);
        //call the copy constructor to initialize an
        //entirely new CStr object with the first
        //operand
  new_string.cat(str2.get_Data());
        //concatenate the second operand onto the
        //end of new_string
  return new_string;
        //call copy constructor to create a copy of
        //the return value new_string
}
```

new_string

strcat(str1,str2)
strlen(str1)+ strlen(str2)

How does it work?

```
CStr first("John");
CStr last("Johnson");
CStr name(first+last);
```



Implementing Operator Overloading

- **Two ways:**
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data

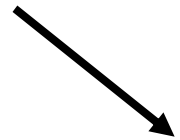
- **Expression *obj1@obj2* translates into a function call**
  - *obj1.operator@(obj2)*, if this function is defined within class obj1
  - *operator@(obj1,obj2)*, if this function is defined outside the class obj1

Implementing Operator Overloading

1. Defined as a member function

```
class Complex {
   ...
 public:
   ...
   Complex operator +(const Complex &op)
   {
     double real   = _real   + op._real,
                 imag = _imag + op._imag;
     return(Complex(real, imag));
   }
   ...
 };
```

```
c = a+ b;
```



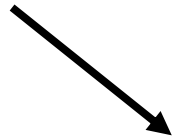
```
c = a.operator+  (b);
```

Implementing Operator Overloading

2. Defined as a non-member function

```
class Complex {
   ...
 public:
   ...
  double real() { return _real; }
    //need access functions
  double imag() { return _imag; }
   ...
 };
```

```
c = a+ b;
```



```
c = operator+  (a, b);
```

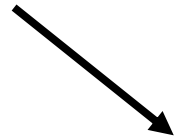
```
Complex operator +(Complex &op1, Complex &op2)
{
   double real   = op1.real()   + op2.real(),
                imag = op1.imag() + op2.imag();
   return(Complex(real, imag));
}
```

Implementing Operator Overloading

3. Defined as a friend function

```
class Complex {
   ...
 public:
   ...
  friend Complex operator +(
     const Complex &,
     const Complex &
   );
   ...
 };
```

```
c = a+ b;
```



```
c = operator+  (a, b);
```

```
Complex operator +(Complex &op1, Complex &op2)
{
   double real   = op1._real   + op2._real,
                 imag = op1._imag + op2._imag;
   return(Complex(real, imag));
}
```