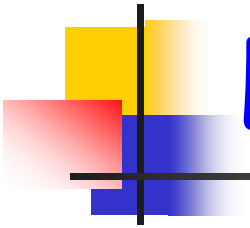
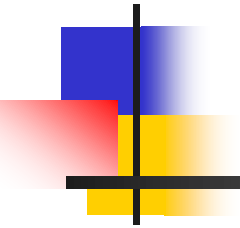
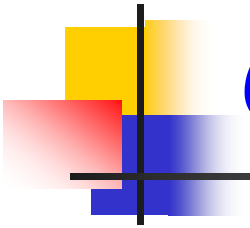


Lecture-6

- Operator overloading
- Namespaces
- Standard template library
  - vector
  - List
  - Map
  - Set
- Casting in C++



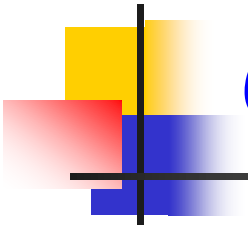
Operator Overloading



Operator overloading

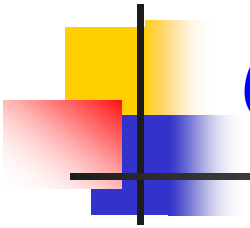
- On two objects of the same class, can we perform typical operations like
  - Assignment (=), increment (++), decrement(--)
  - Write to a stream (<<)
  - Reading from a stream (>>)
- Can be defined for user defined classes.

  ⇒ Operator overloading
- Most of the common operators can be overloaded.
- Operators – can be member/non-member functions



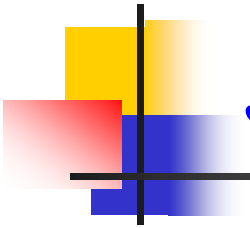
Operator overloading … cont.

- Arity of operator
  - Number of parameters required.
- Unary operators – take one argument
  - *E.g.*, ++, --, !, ~, etc.
  - C unary operators remain unary in C++
- Binary operators – take two arguments.
  - *E.g.*, =, >, <, +, -, etc.
  - C binary operators remain binary.
- Typical overloaded operators
  - +, -, >, <, ++, --, +=, ==, !=, <=, >=, <<, >>, [ ]



Operator functions rules

- Member function operators
  - Leftmost operand must be an object (or reference to an object) of the class.
  - If left operand is of a different type, operator function must NOT be a member function
- Built-in operators with built-in data types CANNOT be changed.
- Non-member operator function must be a **`friend`** if
  - **`private`** or **`protected`** members of that class are accessed directly



Syntax

- Member function

```
return_type classname :: operator symbol (args)
  {
    // code
  }
```

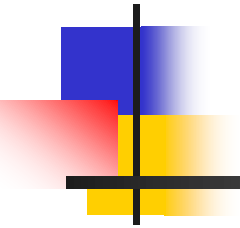
- Non-Member function

```
return_type operator symbol (args)
  {
    // code
  }
```

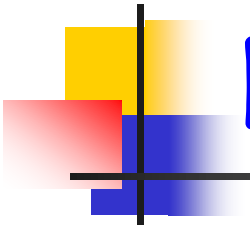
Example

```
class Integer
{
private:
  int value;
public:
  Integer (int val) : value (val) {  }
  void operator++( ) { value++; }              // Member op
  void operator-( ) { value = 0 - value; }  // Member op
   // Non-member friend function operator
  friend Integer operator +  (const Integer& i, const Integer& j);
  void showValue( ) { cout << "Value is: " << this->value << endl; }
};

Integer operator + (const Integer&i, const Integer& j)
{
  return Integer (i.value + j.value);
}
```



Namespaces

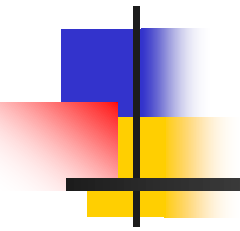


Namespaces

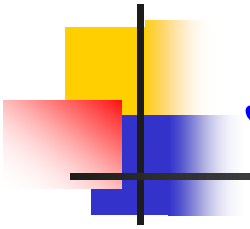
- A way to give "scope" to different variables, as opposed to a global scope.
  - E.g., namespace myNameSpace
    { int a, b; }
  - a and b are visible ONLY in myNameSpace
  - Can be accessed using the keywork "using"
    - using namespace myNamespace
      a = 3; b = 4;
    - myNamespace::a = 3; myNamespace::b = 4;
    - using myNamespace::a
      cout << "value of a is: " << a << endl;

Namespace example

```
namespace myNameSpace
{
   int a = 3;
   double b = 4.3;
   void printHello ( ) { cout << "Hello" << endl; }
}
int main ( )
{
    using namespace myNameSpace;
    printHello( );
    // Note: No declaration for a and b
   cout << "a: " << a << << " b: " << b << endl;
}
```



Standard Template Libraries (STL)

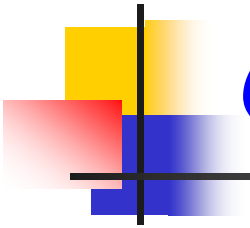


Standard template library

- Defines many useful classes.
- Popular among them
  - vector, list, map, set, iterators, etc.
  - Each of these is a template class.
  - Has many useful functions.
- References:

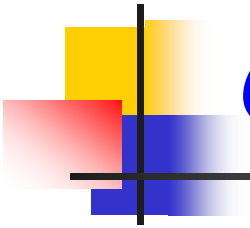
http://www.cplusplus.com/reference/stl/

They list all the functions, coding examples and many nice features for vectors, lists and iterators.



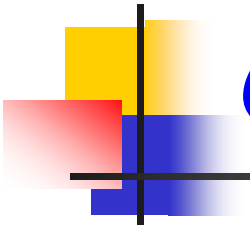
C++ type casting

- Casting: converting a variable/object of one type/class to another.
- C type casting can create run time errors in C++.
- C++ needs to cast between "related" classes
- C++ provides additional casting methods
  - dynamic_cast
  - static_cast
  - reinterpret_cast
  - const_cast



C++ type casting … contd.

- E.g., Base b, *bp; Derived d, *dp
- dynamic_cast:
  - Casting from derived to base class, NOT from base to derived to base class.
    - bp = dynamic_cast <base*> (dp); // Allowed
    - dp = dynamic_cast <derived *>(b); // NOT allowed
- static_cast:
  - Casting from base to derived and vice-versa.
    - bp = static_cast <base*> (d); // Allowed
    - dp = static_cast <derived *>(b); // allowed



Casting … contd.

- reinterpret_cast:
  - Binary copy of values from one pointer to another
  - Can cast any type to any other type, even for unrelated class objects.
    - `Class A {.. };  Class B {..}; // not related to A`

      `A *pa = new A;`

      `B *pb = reinterpret_cast<B*> pa;`

    Suggestion: Don't use it unless you know what you are doing.
- const_cast:
  - Set or remove the const'ness of an object
  - Const object can be passed as an non-const argument to a function.

const_cast - example

```
#include <iostream>
using namespace std;
void myPrint (char * str)
{
   cout << str << endl;
 }

int main ( )
{
   const char * c = "sample text";
   myPrint ( const_cast<char *> (c) );
   return 0;
}
```