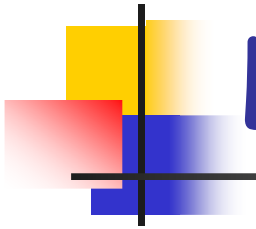


Lecture-4

- Templates
- Operator overloading

Templates – motivation

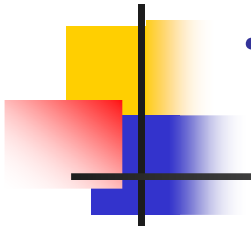
- Consider a function to sort integers

```
int findMaxInt (int num1, int num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- Consider a function to sort floats

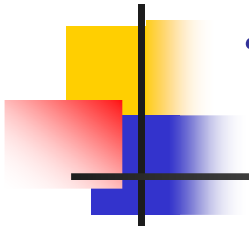
```
float findMaxFloat (float num1, float num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- Same code, but two functions are needed
  - One for each data type



Templates motivation … cont.

- Problem here
  - One max. function is needed for each data type.
  - But the code finding max itself is the same.
- Templates are used to solve this issue.
- Templates
  - Define functions with generic types
  - Define generic classes.
  - Same code can be used with any data type.
  - No need for code repetition.



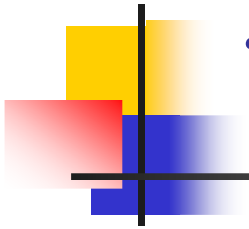
Template functions

- Define functions with generic types

```
template <class anyType>
        function definition
```

- Example

```
template <class anyType>
    anyType GetMax (anyType a, anyType b)
    {
            if (a > b) return a;
            return b;
    }
```



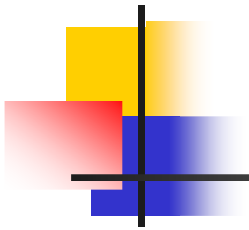
Template classes

- Define any class
- Use template definition to define any generic data type.
- Rest of the code remains the same.

```
template <class anyType>
class someClass
{
    someClass (…) // constructor
    ~someClass( ) // destructor
//    member data & methods // can use anyType
};
```

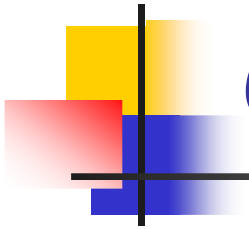
Template class example

```
template <class anyType>
class myClass
{
   private:
     anyType value1, value2;

    public:
       myClass (anyType i, anyType j) : value1(i), value2(j)
   {  }
       ~myClass() { }
       anyType findMaxValue ()
       {
            if (value1 > value2) return (value1);
            return (value2);
       }
};
```



Template class example … contd.

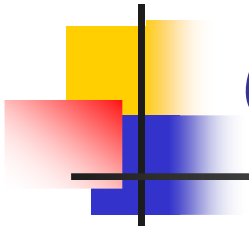
```
main()
{
   myClass <int> intObject (2, 3);
   int maxInt = intObject.findMaxValue();
   cout << "maxInt: " << maxInt << endl;

   myClass <float> floatObject (4.3, 3.2);
   float maxFloat = floatObject.findMaxValue();
   cout << "maxFloat: " << maxFloat << endl;
}
```



Operator overloading

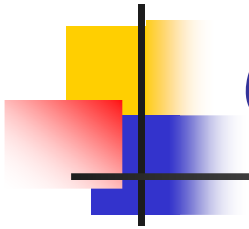
- On two objects of the same class, can we perform typical operations like
  - Assignment (=), increment (++), decrement(--)
  - Write to a stream (<<)
  - Reading from a stream (>>)
- Can be defined for user defined classes.

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



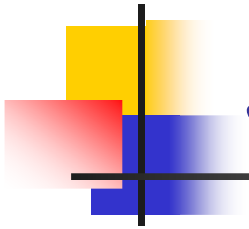
Operator overloading … cont.

- 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



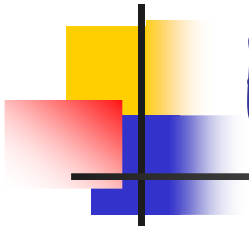
Syntax

- Member function

```
return_type classname :: operatorSymbol (args)
  {
    // code
  }
```

- Non-member function

```
return_type operatorSymbol (args)
  {
    // code
  }
```



Example

```
class Integer
   {
        private:
           int value;
        public:
           Integer (int val) : value (val) {  }
           void operator ++( ) { value++; }  // Member op
           friend Integer operator +      // Non-member op
                (const Integer& i, const Integer& j);
   };

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