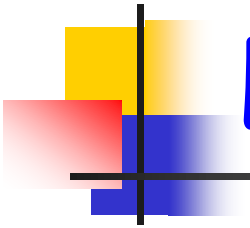
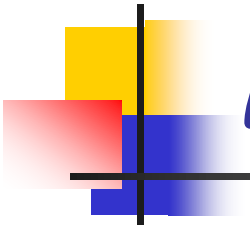


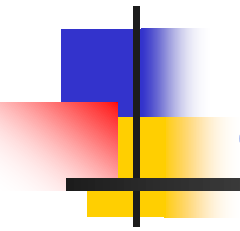
Lecture-5

- Miscellaneous topics
- Templates



Miscellaneous topics

- Miscellaneous topics
  - const member functions, const arguments
  - Function overloading and function overriding
  - this keyword
  - C++ static members
  - inline functions
  - Passing arguments to functions by value, pointer and reference



const arguments and const member functions

const arguments to functions

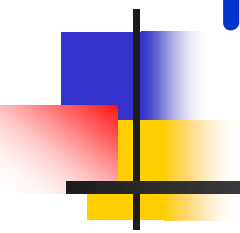
```
void f1(const int a)
{
    a = 3; // Not allowed
}
```

- const arguments to a function can't be changed in the function.
- f1 can't change a in the above example

const member functions

```
class myClass
{
   int a;
   …
   void f1( ) const
    { a = 3; } // Not allowed
};
```

- const functions can't change any attributes of myClass.
- f1 can't change a in the above example



Function overloading and function overriding

Function overloading

- Functions with the same name but with
  - Different number of arguments or
  - Different types of arguments

E.g. int add (int a, int b, int c) { return (a+b+c); }
int add (int a, int b) { return (a + b); }
double add (double a, double b) { return (a + b); }

- Here "add" is an overloaded function

Function overriding

- Functions defined in parent class and re-implemented by the child class.

E.g. class Bird

```
{
    int canFly( ) { return (1); }
}
class Penguin : public Bird
{
   int canFly ( ) { return (0); }
}
```

Here, "canFly" is an overridden by the child class, Penguin

this keyword

- this refers to the addrss of the current object
- E.g.

```
class Account
{
   private:
      int balance;
   public setBalance (int amount)
    {
       this->balance = amount;
    }
};
```

C++ static members

- static members in C++
  - Shared by all the objects of a class
  - Specific to a class, NOT object of a class
  - Access them using className::static_member
  - E.g., myClass::staticVar, or myClass::f1
  - NOT myClassObj.staticVar or myClassObj.f1( )

```
class myClass
{
  public:
        static int staticVar;
        static void f1( );
};
```

Inline functions … contd.

- Compiler decides if a function defined as "inline" can be "inline" or not.
- Too much of code for any function defined as inline
  - Compiler may treat as a regular (non-inline) function
- Note: Code for an inline function
  - Can be in the class itself, or
  - Can be in the same file as the class definition.
  - CANNOT be defined in any file outside the class definition

Inline functions

- Normal functions
  - Carry operational overhead
  - Function call, parameter passing, etc.
- Inline functions
  - No overhead related to function calls
  - Might be as simple as a memory access

```
class myClass
{
        public:
                int x;
                inline int getX ( ) { return x; }
};
```

Passing args. to a function … by value

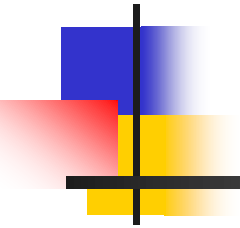
- Compiler creates its own copy.
- Any changes made inside the function are not reflected after the function.

```
class myClass
{
        void f1(int  i ) // i is passed by value
         { i = 3; }
};
int x = 5;
myClass obj;
obj.f1( x );
cout << "value of x: " << x << endl; // x is still 5
```

Passing args. to a function … by reference

- Compiler takes the original object.
- Any changes made inside the function are reflected after the function.

```
class myClass
{
        void f1(int& i ) // i is passed by reference.
         { i = 3; }
};
int x = 5;
myClass obj;
obj.f1( x );
cout << "value of x: " << x << endl; // x is 3
```



Templates

Templates – motivation

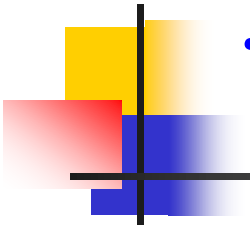
- Consider a function to sort integers

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

- Consider a function to sort floats

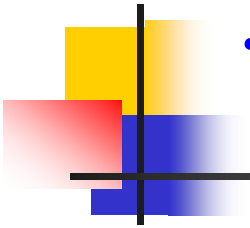
```
float findMax (float num1, float num2)
{
        if (num1 > num2) return num1;
        return num2;
}
```

- 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
{
    anyType a;
    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;
}
```