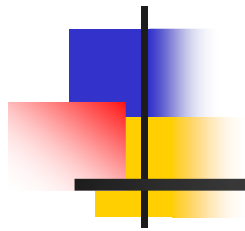
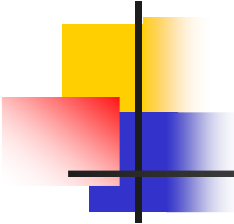


Lecture-5

- Templates - review
- Standard template library
 - vector
 - list
- C++ **static** members
- **this** keyword
- Inline functions
- Operator overloading
- Casting in C++
- Namespaces



Templates

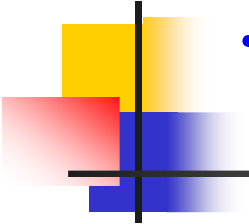


Templates - motivation

- Consider a function to sort integers

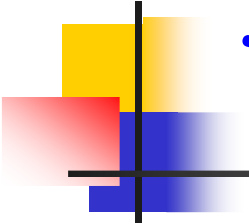
```
int findMaxInt (int num1, int num2)
{
    if (num1 > num2) return num1;
    return num2;
}
```
- Consider a function to sort floats

```
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>
```

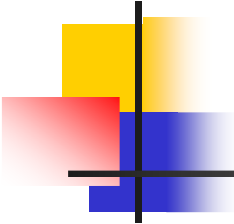
```
anyType GetMax (anyType a, anyType b)
```

```
{
```

```
    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.

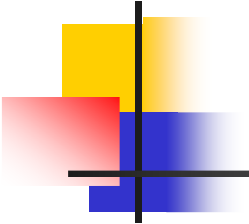
```
template <class anyType>
class someClass
{
    anyType a;
    someClass (...) // constructor
    ~someClass( ) // destructor
    // member data & methods // can use anyType
};
```



Template class example

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

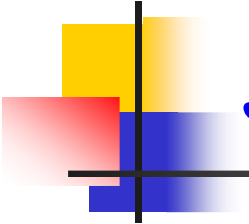
    public:
        myClass (anyType i, anyType j) : value1(i), value2(j)
        { }
        ~myClass() { }
        anyType findMaxValue ()
        {
            if (value1 > value2) return (value1);
            return (value2);
        }
};
```



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;
}
```

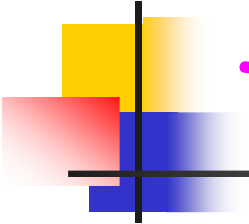


Standard template library

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

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

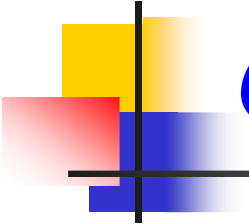
They list all the functions, coding examples and many nice features for vectors, lists and iterators.



this keyword

- **this** refers to the address of the current object
- E.g.

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



C++ static members

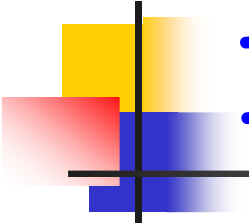
- static members in C++
 - Shared by all the objects of a class
 - Specific to a class, **NOT** object of a class
 - Access them using `className::static_member`
 - E.g., `myClass::staticVar`, or `myClass::f1`
 - **NOT** `myClassObj.staticVar` or `myClassObj.f1( )`

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



Inline functions ... contd.

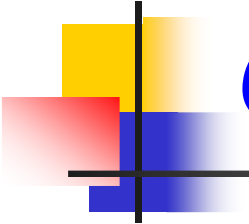
- Compiler decides if a function defined as "inline" can be "inline" or not.
- Too much of code for any function defined as inline
 - Compiler may treat as a regular (non-inline) function
- Note: Code for an inline function
 - Can be in the class itself, or
 - Can be in the same file as the class definition.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



Inline functions

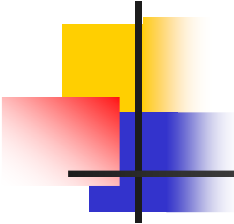
- Normal functions
 - Carry operational overhead
 - Function call, parameter passing, etc.
- Inline functions
 - No overhead related to function calls
 - Might be as simple as a memory access

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



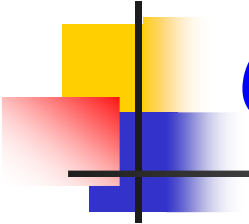
Operator overloading

- On two objects of the same class, can we perform typical operations like
 - Assignment (=), increment (++), decrement(--)
 - Write to a stream (<<)
 - Reading to 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



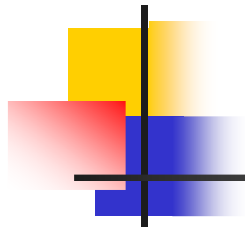
Operator overloading ... cont.

- 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



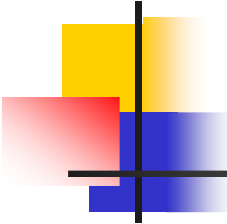
Syntax

- Member function

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

- Non-Member function

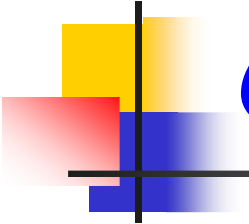
```
return_type operator symbol (args)
{
    // code
}
```



Example

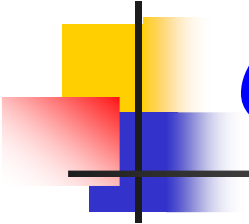
```
class Integer
{
private:
    int value;
public:
    Integer (int val) : value (val) { }
    void operator++( ) { value++; }           // Member op
    void operator--( ) { value = 0 - value; } // Member op
    // Non-member friend function operator
    friend Integer operator + (const Integer& i, 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);
}
```



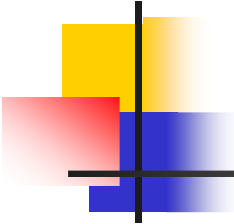
C++ type casting

- Casting: converting a variable/object of one type/cast to another.
- C type casting can create run time errors in C++.
- C++ needs to casting 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*> (d); // 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`

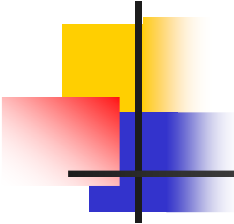
- `A *pa = new A;`

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

- Suggestion: Don't use it unless you know what you are doing.**

- `const_cast`:

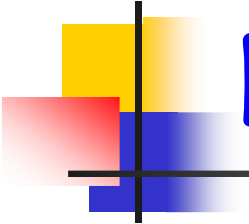
- Set or remove the `const`'ness of an object
- Const object can be passed as an non-const argument to a function.



const_cast - example

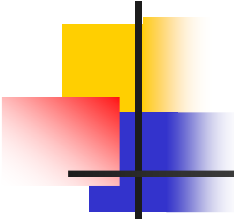
```
#include <iostream>
using namespace std;
void myPrint (char * str)
{
    cout << str << endl;
}
```

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



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 keyword **"using"**
 - using namespace myNamesapce
 a = 3; b = 4;
 - myNamesapce::a = 3; myNamesapce::b = 4;
 - using myNamesapce::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;
}
```