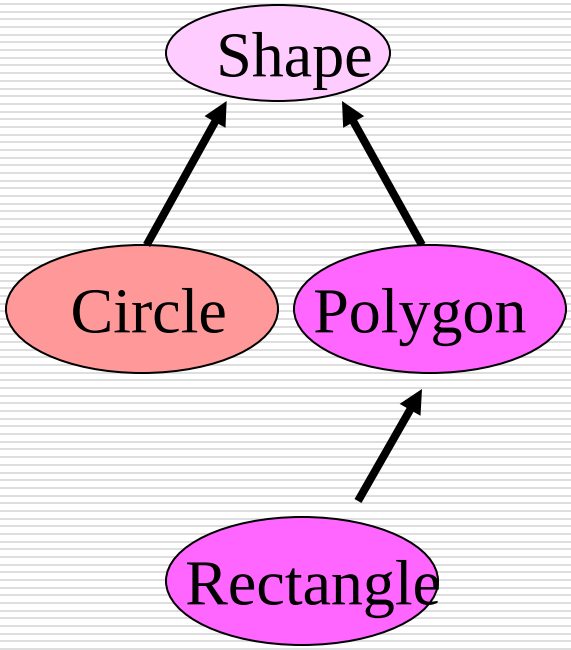
Lecture 7

Abstract Class

- An abstract class represents an abstract concept in C++ (such as Shape class)



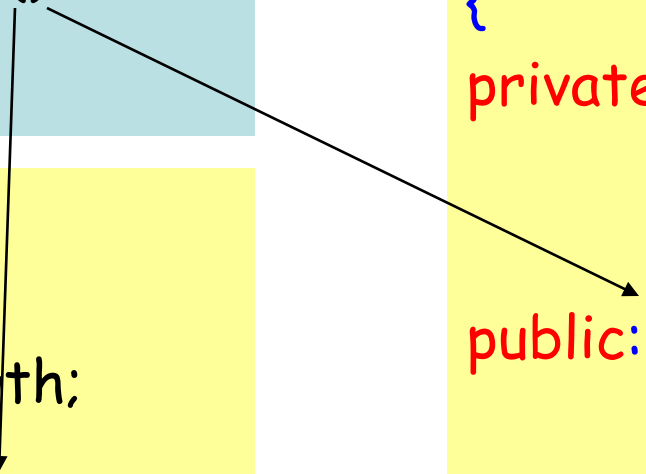
1. Defines the interfaces that all of the concrete classes (subclasses) share
2. Does not define state and implementation unless it is common to all concrete classes
3. Cannot be instantiated

Virtual functions

```
class shape
{
public:
    virtual void area() = 0;
};
```

```
class Triangle{
    protected:
        int width, length;
    public:
            void area() {…};
}
```

```
class Circle : public
    Point
{
private:
            int x, y;
            double r;
public:
            void area() {…};
}
```



More C++ Concepts

- Operator overloading
- Friend Function
- This Operator
- Inline Function

Review

- There are different types of member functions in the definition of a class
  - accessor
    - int CStr :: get_length();
  - implementor/worker
    - void Rectangle :: set(int, int);
  - helper
    - void Date :: errmsg(const char* msg);
  - constructor
    - Account :: Account();
    - Account :: Account(const Account& a);
    - Account :: Account(const char *person);
  - destructor
    - Account :: ~Account();

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

// integer addition and assignment
k = i + j;

// floating addition and assignment
p = m + n;
```

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

```
Date 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);
    // …
    friend CStr operator+(CStr str1,
    CStr str2);
    friend CStr operator+(CStr str, char
    *s);
    friend 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

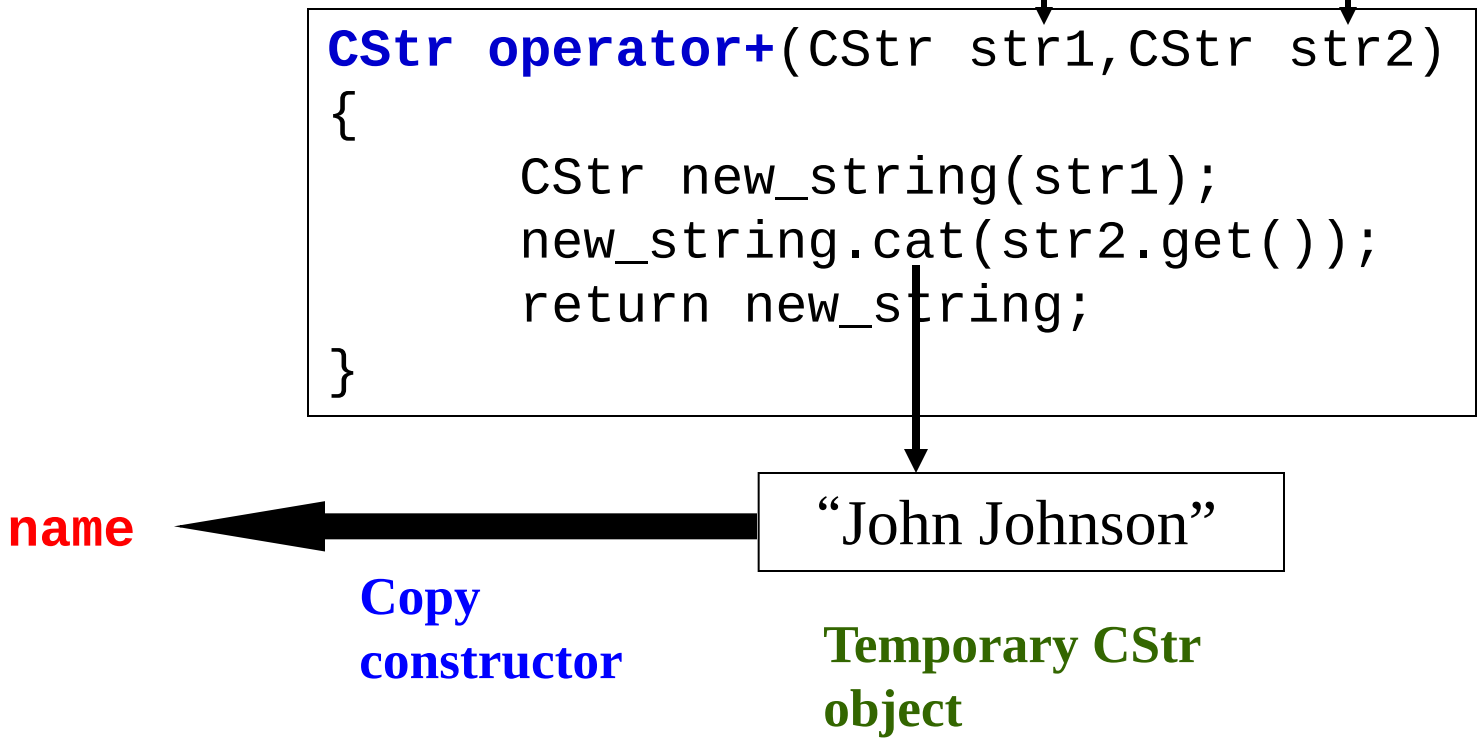
```
CStr operator+(CStr str1, CStr str2)
{
  CStr new_string(str1);
        //call the copy constructor to
   initialize an  entirely new CStr object
   with the first          operand
  new_string.cat(str2.get_Data());
        //concatenate the second operand
   onto the      end of new_string
  return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

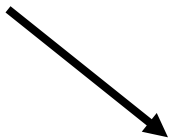
- Two ways:
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data
- Expression *obj1@obj2* translates into a function call
  - *obj1.operator@(obj2)*, if 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;
```

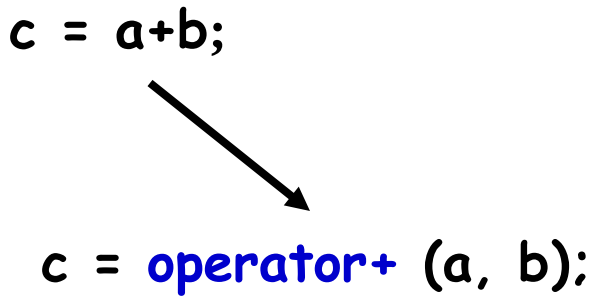


```
c = a.operator+ (b);
```

Implementing Operator Overloading

2. Defined as a non member function

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



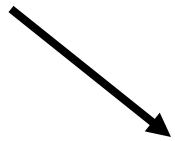

```
Complex operator +(Complex &op1, Complex &op2)
{
    double real   = op1.real()   + op2.real(),
                  imag = op1.imag() + op2.imag();
    return(Complex(real, imag));
}
```

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));
}
```

What is “Friend”

- Friend declarations introduce extra coupling between classes
  - Once an object is declared as a friend, it has access to all non-public members as if they were public
- Access is unidirectional
  - If B is designated as friend of A, B can access A’s non-public members; A cannot access B’s
- A friend function of a class is defined outside of that class's scope

More about “Friend”

- The major use of friends is
  - to provide more efficient access to data members than the function call
  - to accommodate operator functions with easy access to private data members
- Friends can have access to everything, which defeats data hiding, so use them carefully
- Friends have permission to change the internal state from outside the class. Always recommend use member functions instead of friends to change state

Assignment Operator

- Assignment between objects of the same type is always supported
  - the compiler supplies a hidden assignment function if you don't write your own one
  - same problem as with the copy constructor - the member by member copying
  - Syntax:

```
class& class::operator=(const class &arg)
{
                    //…
}
```

Example: Assignment for CStr class

Assignment operator for CStr:

```
CStr&    operator=(const CStr & source)
```

Return type - a reference to (address of) a CStr object

Argument type - a reference to a CStr object (since it is const, the function cannot modify it)

```
CStr& operator=(const CStr &source){
//... Do the copying
return *this;
}
```

Assignment function is called as a member function of the left operand =>Return the object itself

```
str1=str2;

str1.operator=(str2)
```

Overloading Stream-insertion and Stream-extraction Operators

- In fact, cout<< or cin>> are operator overloading built in C++ standard lib of *iostream.h*, using operator "<<" and ">>"
- cout and cin are the objects of ostream and istream classes, respectively
- We can add a friend function which overloads the operator <<

```
friend ostream& operator<< (ostream &os, const Date &d);
```

```
ostream& operator<<(ostream &os, const Date &d)
{
   os<<d.month<<"/"<<d.day<<"/"<<d.year;
   return os;
}
...
cout<< d1;  //overloaded operator
```

cout ---- object of ostream

Overloading Stream-insertion and Stream-extraction Operators

- We can also add a friend function which overloads the operator >>

```
friend ostream& operator >> (ostream &in, const Date &d);
```

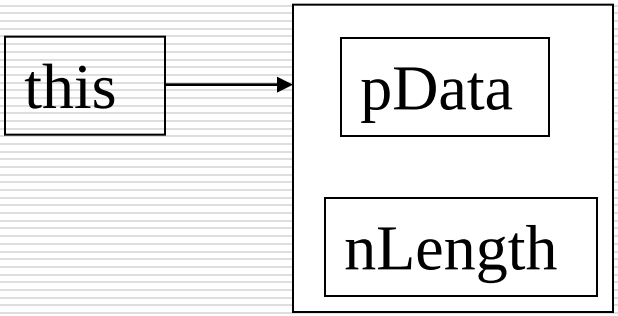
```
istream& operator>> (istream &in, Date &d)
{
   char mmddyy[9];
   in >> mmddyy;
   // check if valid data entered
   if (d.set(mmddyy))   return in;
   cout<< "Invalid date format: "<<d<<endl;
   exit(-1);
 }
```

cin ---- object of istream

```
cin >> d1;
```

The “this” Operator

- Within a member function, the *this* keyword is a pointer to the current object, i.e. the object through which the function was called
- C++ passes a hidden *this* pointer whenever a member function is called
- Within a member function definition, there is an implicit use of *this* pointer for references to data members



CStr object
(*this)

Data member reference	Equivalent to
pData	this->pData
nLength	this->nLength

Inline Functions

- An inline function is one in which the function code replaces the function call directly.
- Inline class member functions
  - If they are defined as part of the class definition, implicit
  - If they are defined outside of the class definition, explicit, I.e.using the keyword, *inline*.
- Inline functions should be short (preferable one-liners).
  - Why? Because the use of inline function results in duplication of the code of the function for each invocation of the inline function

Example of Inline functions

```
class CStr
{
  char *pData;
  int nLength;
    …
  public:
    …
    char *get_Data(void) {return pData; } //implicit inline function
    int getlength(void);
     …
};

inline void CStr::getlength(void)  //explicit inline function
{
  return nLength;
}
    …

int main(void)
{
  char *s;
  int n;
  CStr a("Joe");
  s = a.get_Data();
  n = b.getlength();
}
```

Inline functions within class declarations

Inline functions outside of class declarations

In both cases, the compiler will insert the code of the functions get_Data() and getlength() instead of generating calls to these functions

Inline Functions – Cont'd

- An inline function can never be located in a run-time library since the actual code is inserted by the compiler and must therefore be known at compile-time.
- It is only useful to implement an inline function when the time which is spent during a function call is long compared to the code in the function.

Take Home Message

- Operator overloading provides convenient notations for object behaviors
- There are three ways to implement operator overloading
  - member functions
  - normal non-member functions
  - friend functions