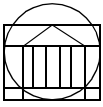


Another Example

```
class Digit
{
    private:
        int value;
    public:
        void print_digit();
        void set(int number);
        int get_val();
};

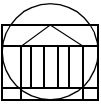
int Digit::get_val()
{
    return value;
}

void Digit::print_digit()
{
    int i ;
    if (value == 0) cout << '0';
    else
    {
        for (i = 1; i <= value; i++)
            cout << 'X';
    }
}
```



```
void Digit::set(int number)
{
    if ((number >= 0) && (number <= 9))
    {
        value = number;
    }
    else
    {
        cout << "Improper value.\n"
              << "Execution terminated.\n";
        exit(1);
    }
    return ;
}

int main()
{
    int integer = 3;
    Digit three;
    three.set(5);
    integer = three.get_val();
    three.print_digit();
    cout << integer;
    cout << three.value;
    return 0;
}
```



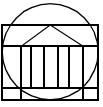
Overloaded Member Functions

- Member function names can be overloaded.

```
class Digit
{
    private:
        int value;
    public:
        void print_digit();
        void print_digit(char new_char);
        void set(int number);
};

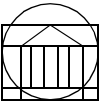
void Digit::print_digit()
{
    int i ;
    if (value == 0) cout << '0';
    else
        for ( i = 1; i <= value; i++ )
            cout << 'X';
}

void Digit::print_digit(char new_char)
{
    int i ;
    if (value == 0) cout << '0';
    else
        for ( i = 1; i <= value; i++ )
            cout << new_char;
}
```



```
int main()  
{  
    int integer = 3;  
    Digit three;  
    three.set(5);  
    three.print_digit();  
    cout << endl;  
    three.set(integer);  
    three.print_digit('A');  
    return 0;  
}
```

```
XXXXX  
AAA
```

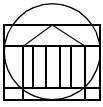


I/O Stream Arguments

- Member function can be defined so that it can be called with an argument that is either `cin/cout` or file I/O stream.

```
class Digit
{
    private:
        int value;
    public:
        void print_digit(ofstream& sout);
        void set(int number);
};

void Digit::print_digit(ofstream& sout)
{
    int i ;
    if (value == 0) sout << '0';
    else
        for ( i = 1; i <= value; i++ )
            sout << 'X';
}
```

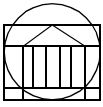


```
#include <iostream.h>
#include <fstream.h>

int main()
{
    Digit one;

    char FILENM[] = "second.out" ;
    ofstream fout ;
    fout.open(FILENM) ;

    one.set(1);
    one.print_digit(cout);
    one.print_digit(fout);
    return 0 ;
}
```



Summary of Properties of Classes

In a class definition:

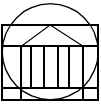
- A class may use another class as the type for a member
- A member (variable or function) can be public or private
- Usually, all member variables are labeled as private
- The name of a member function can be overloaded
-

For each object declared to be of a class type:

- Can have both member variables and member functions
- A private member *can only* be used by another member of the same object
-

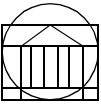
In function definitions:

- A function can have formal parameters whose types are classes
- A class may be the type for the value returned by a function



Constructors

- It is often convenient to initialize some or all the member variables for an object when the *object is declared*.
- Constructor: a special kind of member function that is automatically called when an object of the class is declared.
 - constructor is used for initialization
 - constructor must have the same name as the class
 - constructor definition cannot return a value
 - no type, not even **void**, can be given in the constructor function prototype or in the constructor function header
 - constructor cannot be called in the same way as other member functions are called
 - constructor must be **public**



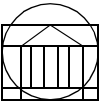
```
class Automobile
{
  public:
    Automobile (int n_year, double n_price);
    void print();
    void set(int new_year, double new_price);
    int get_year();
    double get_price();
  private:
    int year;
    double price;
};
```

```
Automobile::Automobile(int n_year,
                       double n_price)
{
  year = n_year;
  price = n_price;
}
```

```
Automobile mine(1997, 9000.00); // fine
```

```
Automobile yours;
yours.Automobile(1997, 9000.00); //Illegal!
```

(cannot "call" a constructor like a member function)

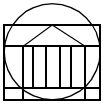


Overloading a Constructor

- A constructor can be overloaded like any other function so that an object can be initialized in more than one way.
 - be careful when an overloaded constructor is used

```
class Automobile
{
  public:
    Automobile (int n_year, double
n_price);
    Automobile();
    void print();
    void set(int new_year, double
new_price);
    int get_year();
    double get_price();
  private:
    int year;
    double price;
};
```

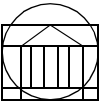
Now we define the two constructors



```
Automobile::Automobile(int n_year,  
                        double n_price)  
{  
    year = n_year;  
    price = n_price;  
}
```

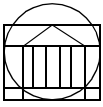
```
Automobile::Automobile()  
{  
    year = 1997;  
    price = 10000.00;  
}
```

```
Automobile mine(1997, 9000.00);  
Automobile yours;  
Automobile theirs();//Illegal!  
yours.Automobile(1997, 9000.00); //  
    Illegal!
```



Another Example

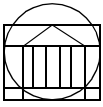
```
class Digit
{
    private:
        int value;
    public:
        Digit();
        Digit(int init);
        void print_digit();
        int getval();
};
Digit::Digit()
{
    value = 5;
}
Digit::Digit(int init)
{
    if ((init >= 0) && (init <= 9))
        value = init;
    else
    {
        cout << "Invalid
initialization.\n";
        exit(1);
    }
}
```



```
void Digit::print_digit()
{   int i ;
    if (value == 0) cout << '0';
    else
        for ( i = 1; i <= value; i++ )
            cout << 'X';
    cout << endl;
    return ;
}

int main()
{
    int one = 1;
    Digit three, XXX(one), zero(0);
    one = three.getval();
    cout << "The value is " << one << endl;
    XXX.print_digit();
    zero.print_digit();
    return 0 ;
}
```

```
The value is 5
X
0
```



Overloading an Operator

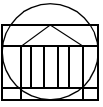
An operator can be overloaded so that it will accept arguments of a class type, in a similar way that a member function can be overloaded.

must use the key word **operator**

```
class Digit
{
public:
    Digit();
    Digit(int init);
    void set(int number);
    void print_digit();
    int getval();
private:
    int value;
}; // end class definition

Digit operator +(Digit a, Digit b)
{
    int answer;
    Digit result;
    answer = a.getval() + b.getval();
    result.set(answer);
    return (result);
}
```

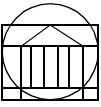
Note: possible because of accessor function `get_val()`



```
void Digit::set(int number)
{
    if ((number >= 0) && (number <= 9))
    {
        value = number;
    }
    else
    {
        cout << "Improper value.\n"
              << "Execution terminated.\n";
        exit(1);
    }
    return ;
}

int main()
{
    int integer = 3;
    Digit three(integer), five(5), what(1);
    what.print_digit();
    what = three + five;
    what.print_digit();
    return 0;
}
```

This outputs.....



With No Accessor Function

Suppose we didn't have member function `int get_val()` in the class definition????

```
Digit operator +(Digit a, Digit b)
{
    int answer;
    Digit result;
    // WHY NOT:????
    answer = a.value + b.value;
    result.set(answer);
    return (result);
}
```

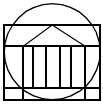
ANSWER: because value is private to Digit.

But CAN do this if you make + a "friend" of class Digit...

```
class Digit
{
public:
    Digit();
    Digit(int init);
    void set(int number);
    void print_digit();
    // int getval();..... remember, eliminated this
    friend Digit operator +(Digit a, Digit b);

private:
    int value;
}; // end class definition
```

(see Savitch, pp. 425-437, 445-448)



One Last Example

Suppose we have:

```
Digit number_of_sticks(3);  
cout << number_of_sticks;
```

But, we can't do this, because the insertion operator, <<, is only defined for two operands of types (ofstream, int(etc))

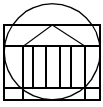
SO, we need to overload the << operator. Do this as an "auxilliary function".

Think: `cout << digit1 << digit2 << endl;`

must be possible..... SO

the result of this "operation" must be an ofstream reference

```
ofstream& operator << (ofstream& out, const Digit& d);
```



So, let's write the function definition:

```
ofstream& operator << (ofstream& out,  
    const Digit& d)  
{  
    int intval;  
    intval = d.getval();  
    out << intval;  
    return out;  
}
```

OR

```
ofstream& operator << (ofstream& out,  
    const Digit& d)  
{  
    int intval;  
    intval = d.getval();  
    if (intval == 0) out << '0';  
    else  
    {  
        for int i = 1; i <= intval; i++)  
            out << 'X';  
    }  
    return out;  
}
```