

# Object-Oriented Programming

- In C++ classes provide the functionality necessary to use *object-oriented programming*
  - OOP is a particular way of organizing computer programs
  - It doesn't allow you to do anything you couldn't already do, but it makes it arguably more efficient
  - OOP is by far the dominant software engineering practice in the last two decades
- Classes combine data and functionality
  - Class members can store structured data, as we've seen
  - Class members can also be functions
    - Class-specific functions are called *methods*

# The string class

- The string class has private data members to store the characters that make up a string
  - It probably uses an array, although it doesn't have to
  - It probably has ints to keep track of the size of the array and the number of characters
- The string class has public methods to do stuff
  - Return the number of characters in the internal storage  
`int len();`
  - Append the characters in s to the internal storage  
`void append( string s );`
  - returns the position of s within the internal storage  
`int find( string s );`

# Date class

- What data should the Date class store?

# Date class

- What data should the Date class store?

```
class Date
{
public:
    int day, mon, yr; // members
};
```

- What functionality would we like Dates to have?

# Printing a date

- We'd like to have a print method so we could do:

```
Date my_birthday; // define an object
my_birthday.yr = 1975;
my_birthday.mon = 5;
my_birthday.day = 15;

my_birthday.Print();
```

- And have it print out “5/15/1975” or “May 15, 1975”
- Notice that the Print() method *is called on the object* my\_birthday
  - We want it to *print the values stored in that object*

# Printing a date

- To do this, we *declare* a **method** in the Date class
  - A method is a class member that is a function

```
class Date
{
public:
    int day, mon, yr;
    void Print(); // declare a function inside of the class Date
};
```

- Data members look like variable declarations
- Method declarations look like function prototypes
- Like a prototype, a method declaration tells the compiler to expect us to define a function later

# Printing a date

- Defining a method looks just like defining a function

```
class Date
{
public:
    int day, mon, yr;
    void print_func(); // declare a function ⇔ method
};

// define a function under a class.
void Date:: print_func()
{
    cout << mon << "/" << day << "/" << yr;
}
```

The name of the method must be *fully qualified*

- <class name>::<name> (e.g. Date::print\_func)
- :: is the **scope resolution operator**

# Variable scope

- In a function, you can use variables that are:
  - Locally declared
  - Declared as a parameter

```
void print( int mon, int day, int yr )  
{ // here, mon, day, yr are parameters  
  char sep = '/'; // sep is local variable.  
  cout << mon << sep << day << sep << yr;  
}
```

- These variables are in **scope**
- When the function ends, local vars and params are discarded



# Variable scope

- In a method, you can use variables that are:
  - Locally declared  $\Leftrightarrow$  local variable
  - Declared as a parameter
  - Or **declared as a class member!**  $\Leftrightarrow$  using members

```
void Date::Print()  
{ // mon, day, yr are a class member  
  cout << mon << "/" << day << "/" << yr;  
}
```

- Class variables reference memory in the object that the method is *called on*
  - The method runs **in the scope of** the object
  - These variables are **not discarded when the method ends!**

# Class Initialization

- When we create a Date object

```
Date my_birthday; // an object: my_birthday
```

- The member fields are full of garbage
- It might be nice to have them initialize to zero

- **Can't initialize in the class definition**

```
class Date
{
public:
    int day = 0, mon = 0, yr = 0;
};
```

- Since the definition is a blueprint, there's nowhere to store those numbers yet
- We have to initialize them *after* the memory is allocated

# Class Constructors

- Classes let us do this, by defining a **constructor** method
  - Added like any other method except:
    - No return value (not even void)
    - Named after the class

```
class Date
{
public:
    int day, mon, yr;
    Date(); // constructor which has the same name as the class; initialize by default
};

Date::Date()
{
    day = 0;
    mon = 0;
    yr = 0;
}
```

# Class Constructors

- When an object is declared

e.g. `Date d; // declare / create an object`

1. Memory is allocated
2. The constructor is called and initializes the values

# Constructors With Parameters

- But what if we want to set initial values to something better than just a bunch of 0s?
  - Without parameters to the constructor, we can only set default values (i.e. 0)
  - So we can define additional constructors that take values

```
class Date
{
public:
    int day, mon, yr;
    Date( int init_day, int init_mon, int init_yr );
};

Date::Date( int init_day, int init_mon, int init_yr )
{
    day = init_day;
    mon = init_mon;
    yr = init_yr;
}
```

# Constructors With Parameters

- Now we have options when we declare new Date objects:

```
Date some_day;           // inits all to 0
Date today( 4, 14, 2010 ); // inits to those numbers
```

# Exercise

1. Define a `BankAccount` class with:
  - **data members** `account_number` and `balance`
  - A **constructor** (initialize the value for the members in a class) that takes initial values for all members
  - A public `print_func` **method** (function inside a class) that shows the account number and balance
2. Define the constructor and the `print_func` method
3. Use the `BankAccount` class (define an object) to:
  - Declare a `BankAccount` object for account 98392 with a starting balance of \$5.32
  - Print the `BankAccount` information