CIS 190: C/C++ Programming

Lecture 8
Classes in C++
Outline

• Procedural Programming vs OOP

• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading

• Livecoding
Procedural Programming

• up until now, everything we’ve been doing has been *procedural programming*

• code is divided into multiple procedures
  – procedures operate on data (structures), when given correct number and type of arguments

• examples: PrintTrain(), ReadSingerFile(), DestroyList(), ProcessEvents(), etc.
Object-Oriented Programming

• now that we’re using C++, we can start taking advantage of *object-oriented programming*

• adding OOP to C was one of the driving forces behind the creation of C++ as a language
  – C++’s predecessor was actually called “C with Classes”
Object-Oriented Programming

• in OOP, code and data are combined into a single entity called a **class**
  – each **instance** of a given class is an **object** of that class type

• principles of Object-Oriented Programming
  – encapsulation
  – inheritance
  – polymorphism
OOP: Encapsulation

• **encapsulation** is a form of information hiding and abstraction

• data and functions that act on that data are located in the same place (inside a class)

• *ideal*: separate the interface/implementation so that you can use the former without any knowledge of the latter
OOP: Inheritance

• *inheritance* allows us to create and define new classes from an existing class

• this allows us to re-use code
  – faster implementation time
  – fewer errors
  – easier to maintain/update
OOP: Polymorphism

• **polymorphism** is when a single name can have multiple meanings
  – normally used in conjunction with inheritance

• we’ll look at one form of polymorphism today:
  – overloading functions
Outline

• Procedural Programming vs OOP

• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading

• Livecoding
Example: Date

typedef struct date {
    int month;
    int day;
    int year;
} DATE;
typedef struct date {
    int month;
    int day;
    int year;
} DATE;
Parts of a Struct

typedef struct date {
    int month;
    int day;
    int year;
} DATE;

(name of the struct)

(optional) shorter name via typedef
typedef struct date {
    int month;
    int day;
    int year;
} DATE;

(name of the struct)

(member variables of the structure)

((optional) shorter name via typedef)
Using a Struct

• if we want to print a date using the struct, what should our function prototype be?
  ____ PrintDate(_________);
Using a Struct

• if we want to print a date using the struct, what should our function prototype be?

```c
void PrintDate(DATE day);
```
Using a Struct

• if we want to print a date using the struct, what should our function prototype be?
  void PrintDate(DATE day);

• if we want to change the year of a date, what should our function prototype be?
  _____ ChangeYear(______________________);
Using a Struct

• if we want to print a date using the struct, what should our function prototype be?
  ```c
  void PrintDate(DATE day);
  ```

• if we want to change the year of a date, what should our function prototype be?
  ```c
  void ChangeYear(DATE day, int year);
  ```
Morphing from Struct to Class

typedef struct date {
    int month;
    int day;
    int year;
} DATE;
Morphing from Struct to Class

```c
struct date {
    int month;
    int day;
    int year;
};
```

• remove the `typedef` – we won’t need it for the class
Morphing from Struct to Class

class date {
    int   month;
    int   day;
    int   year;
};

• change struct to class
Morphing from Struct to Class

class Date {
    int month;
    int day;
    int year;
};

• capitalize date – according to the style guide, classes are capitalized, while structs are not
Morphing from Struct to Class

class Date {
    int m_month;
    int m_day;
    int m_year;
};

• add m_ to the variable names – classes are more complicated, this can help prevent confusion about which vars are member vars
Morphing from Struct to Class

class Date {
public:
    int m_month;
    int m_day;
    int m_year;
};

• make the variables public, to be able to access them
  – by default, members of a class are private
Morphing from Struct to Class

class Date {
public:
    int m_month;
    int m_day;
    int m_year;
};

• syntax highlighted colors change
Outline

• Procedural Programming vs OOP
• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading
• Livecoding
Functions in Classes

• unlike structs, classes have **member functions** along with their member variables

• member functions go **inside** the class declaration

• member functions are **called on** an object of that class type
Functions in Classes

• unlike structs, classes have *member functions* along with their member variables

• member functions go *inside* the class declaration

• member functions are *called on* an object of that class type

  `iStream.open("file.txt");`
Functions in Classes

• unlike structs, classes have *member functions* along with their member variables

• member functions go **inside** the class declaration

• member functions are **called on** an object of that class type

```cpp
iStream.open(“file.txt”);
```
Example: OutputMonth() Function

• let’s add a function to the class that will print out the name of the month

```cpp
class Date {
public:
    int m_month;
    int m_day;
    int m_year;
};
```
Example: OutputMonth() Function

• let’s add a function to the class that will print out the name of the month

```cpp
class Date {
public:
    int m_month;
    int m_day;
    int m_year;
    void OutputMonth();
};
```
Example: OutputMonth() Function

- let’s add a function to the class that will print out the name of the month, given the number

```cpp
class Date {
public:
    int m_month;
    int m_day;
    int m_year;
    void OutputMonth();
};
```
void OutputMonth();

• nothing is passed in to the function – why?
OutputMonth() Prototype

```c++
void OutputMonth();
```

- nothing is passed in to the function
- because it only needs access to see the variable `m_month`
  - which is a *member variable* of the Date class
  - just like `OutputMonth()` is a *member function*
OutputMonth() Definition

```cpp
void Date::OutputMonth() {
}
```
OutputMonth() Definition

```cpp
void Date::OutputMonth() {
    // specify class name; more than one class can have a function with the same name
}
```
void Date::OutputMonth() {
    this double colon is called the *scope resolution operator*, and associates the *member function* OutputMonth() with the class *Date*.
}
OutputMonth() Definition

```cpp
void Date::OutputMonth() {
    switch (m_month) {
        case 1: cout << "January "; break;
        case 2: cout << "February "; break;
        case 3: cout << "March "; break;
        /* etc */
        default:
            cout << "Error in Date::OutputMonth";
    }
}
```
OutputMonth() Definition

```cpp
void Date::OutputMonth() {
    switch (m_month) {
    case 1: cout << "January"; break;
    case 2: cout << "February"; break;
    case 3: cout << "March"; break;
    /* etc */
    default:
        cout << "Error in Date::OutputMonth";
    }
}
```
Print Functions

• is the following valid code?
  
  ```cpp
  cout << today.OutputMonth();
  ```
Print Functions

• is the following valid code?
  
  ```cpp
  cout << today.OutputMonth();
  ```

• no, because `OutputMonth()` returns nothing for `cout` to print
  – if the function returned a string, this would be valid code
Using the Date Class

Date today;


Using the Date Class

Date today;

variable today is an instance of the class Date

it is an object of type Date
Using the Date Class

```cpp
Date today;

cout << "Please enter dates as DD MM YYYY" << endl;

cout << "Please enter today’s date: ";

cin >> today.m_day >> today.m_month >> today.m_year;
```
Using the Date Class

```cpp
Date today;

cout << "Please enter dates as DD MM YYYY" << endl;

cout << "Please enter today's date: ";
cin >> today.m_day >> today.m_month >> today.m_year;
```

when we are not inside the class (as we were in the `OutputMonth()` function) we must use the dot operator to access `today's member variables`
Using the Date Class

```cpp
Date today;

cout << "Please enter dates as DD MM YYYY" << endl;

cout << "Please enter today's date: ";
cin >> today.m_day >> today.m_month >> today.m_year;

cout << "Today's date is ";
today.OutputMonth();
cout << today.m_day << ", " << today.m_year << endl;
```
Using the Date Class

```cpp
Date today;

cout << "Please enter dates as DD MM YYYY" << endl;

// We also use the dot operator to call the member function OutputMonth() on the Date object today

// Again, note that we do not need to pass in the member variable m_month

today.OutputMonth();

// cout << today.m_day << ", " << today.m_year << endl;
```
Outline

• Procedural Programming vs OOP
• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading
• Livecoding
Access Specifiers

• in our definition of the *Date* class, everything was *public* – this is not good practice!

• why?
Access Specifiers

• we have three different options for access specifiers, each with their own role:
  – public
  – private
  – protected

• specify access for members inside the class
Toy Example

class Date {
public:
    int m_month;
private:
    int m_day;
protected:
    int m_year;
};
Using Public, Private, Protected

- **public**
  - anything that has access to a `Date` object also has access to all public member variables and functions

- not normally used for variables; used for most functions

- need to have at least one item be public
Using Public, Private, Protected

• **private**
  – private members variables and functions can only be accessed by *member functions* of the *Date* class; cannot be accessed in main(), etc.

• if not specified, members default to private
  – should specify anyway – good coding practices!
Using Public, Private, Protected

• protected
  – protected member variables and functions can only be accessed by member functions of the Date class, and by member functions of any derived classes
  – (we’ll cover this later)
Access Specifiers for Date Class

class Date {
    public:
        void OutputMonth();
    private:
        int m_month;
        int m_day;
        int m_year;
};
Access Specifiers for Date Class

class Date {
public:
    void OutputMonth();

private:
    int m_month;
    int m_day;
    int m_year;
};
New Member Functions

• now that `m_month, m_day, and m_year` are `private`, how do we give them values, or retrieve those values?
New Member Functions

• now that *m_month*, *m_day*, and *m_year* are *private*, how do we give them values, or retrieve those values?

• write public member functions to provide indirect, controlled access for the user
  – *ideal*: programmer only knows interface (public functions) not implementation (private variables)
Member Function Types

- there are many ways of classifying types, but here’s a few of the basics we’ll use:
  - accessor functions
  - mutator functions
  - auxiliary functions
Member Functions: Accessor

• *convention*: start with `Get`
• allow retrieval of private data members

• examples:
  ```
  int GetMonth();
  int GetDay();
  int GetYear();
  ```
Member Functions: Mutator

• *convention*: start with `Set`
• allow changing the value of a private data member

• examples:
  ```c
  void SetMonth(int m);
  void SetDay(int d);
  void SetYear(int y);
  ```
Member Functions: Auxiliary

• provide support for the operations
  – public if generally called outside function
  – private/protected if only called by member functions

• examples:

  void OutputMonth(); // public
  void IncrementDate(); // private
Access Specifiers for Date Class

class Date {
public:
    void OutputMonth();
    int GetMonth();
    int GetDay();
    int GetYear();
    void SetMonth(int m);
    void SetDay(int d);
    void SetYear(int y);
private:
    int m_month;
    int m_day;
    int m_year;
};
Access Specifiers for Date Class

class Date {
public:
    void OutputMonth();
    int GetMonth();
    int GetDay();
    int GetYear();
    void SetMonth(int m);
    void SetDay (int d);
    void SetYear (int y);
private:
    int m_month;
    int m_day;
    int m_year;
};

for the sake of brevity, we’ll leave out the accessor and mutator functions from now on
Outline

• Procedural Programming vs OOP
• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading
• Livecoding
Constructors

• special member functions used to create (or “construct”) new objects

• automatically called when an object is created
  – implicit: Date today;
  – explicit: Date today(10, 15, 2014);

• initializes the values of all data members
Date Class Constructors

class Date {
public:
    void OutputMonth();
    Date (int m, int d, int y);
private:
    int m_month;
    int m_day;
    int m_year;
};
Date Class Constructors

class Date {
public:
    void OutputMonth();
    Date (int m, int d, int y);
private:
    int m_month;
    int m_day;
    int m_year;
};
class Date {
public:
  void OutputMonth();
  Date (int m, int d, int y);
private:
  int m_month;
  int m_day;
  int m_year;
};
Constructor Definition

`Date::Date (int m, int d, int y)`

{

}

}`
Constructor Definition

```cpp
Date::Date (int m, int d, int y)
{
    m_month = m;
    m_day = d;
    m_year = y;
}
```
Date::Date (int m, int d, int y) 
{
    
    m_month = m;

    m_day = d;

    m_year = y;

}
Constructor Definition

Date::Date (int m, int d, int y)
{
    if (m > 0 && m <= 12) {
        m_month = m;
    } else {
        m_month = 1;
    }
    if (d > 0 && d <= 31) {
        m_day = d;
    } else {
        m_day = 1;
    }
    if (y > 0 && y <= 2100) {
        m_year = y;
    } else {
        m_year = 1;
    }
}
Constructor Definition

Date::Date (int m, int d, int y)
{
    if (m > 0 && m <= 12) {
        m_month = m;
    }
    else {
        m_month = 1;
    }
    if (d > 0 && d <= 31) {
        m_day = d;
    }
    else {
        m_day = 1;
    }
    if (y > 0 && y <= 2100) {
        m_year = y;
    }
    else {
        m_year = 1;
    }
}
Constructor Definition

**Date::Date** (int m, int d, int y)

```cpp
{
    if (m > 0 && m <= 12) {
        m_month = m;
    } else {
        m_month = 1;
    }
    if (d > 0 && d <= 31) {
        m_day = d;
    } else {
        m_day = 1;
    }
    if (y > 0 && y <= 2100) {
        m_year = y;
    } else {
        m_year = 1;
    }
}
```

**is this the best way to handle this?**

**what might be a better solution?**
Constructor Definition

Date::Date (int m, int d, int y)
{
    SetMonth(m);
    SetDay(d);
    SetYear(y);
}

• this allows us to reuse already written code
Outline

• Procedural Programming vs OOP

• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading

• Livecoding
Overloading

• we can define multiple versions of the constructor – we can overload it

• different constructors for:
  – when all values are known
  – when no values are known
  – when some subset of values are known
All Known Values

• have the constructor set user-supplied values

```cpp
Date::Date (int m, int d, int y)
{
    SetMonth(m);
    SetDay(d);
    SetYear(y);
}
```
All Known Values

• have the constructor set user-supplied values

```cpp
Date::Date (int m, int d, int y)
{
    SetMonth (m);
    SetDay (d);
    SetYear (y);
}
```

invoked when constructor is called with all arguments
No Known Values

• have the constructor set all default values

```cpp
Date::Date ()
{
    SetMonth(1);
    SetDay(1);
    SetYear(1);
}
```
No Known Values

• have the constructor set all default values

```cpp
Date::Date() {
    SetMonth(1);
    SetDay(1);
    SetYear(1);
}
```
invoked when constructor is called with no arguments
Some Known Values

• have the constructor set some default values

```cpp
Date::Date (int m, int d)
{
    SetMonth(m);
    SetDay(d);
    SetYear(1);
}
```
Some Known Values

• have the constructor set some default values

```cpp
Date::Date (int m, int d) {
    SetMonth (m) ;
    SetDay (d) ;
    SetYear (1) ;
}
```

invoked when constructor is called with two arguments
Overloaded Date Constructor

- so far we have the following constructors:

```c
Date::Date (int m, int d, int y);
Date::Date (int m, int d);
Date::Date ();
```
Overloaded Date Constructor

• so far we have the following constructors:

```cpp
Date::Date (int m, int d, int y);
Date::Date (int m, int d);
Date::Date ();
```

• would the following be a valid constructor?

```cpp
Date::Date (int m, int y);
```
Avoiding Multiple Constructors

• defining multiple constructors for different sets of known values is a lot of unnecessary code duplication

• we can avoid this by setting default parameters in our constructors
Default Parameters

• in the **function prototype** only, provide default values you want the constructor to use

```
Date (int m , int d ,
     int y );
```
Default Parameters

• in the *function prototype only*, provide default values you want the constructor to use

    Date (int m = 10, int d = 15, int y = 2014);
Default Parameters

- in the *function definition* nothing changes

```cpp
Date::Date (int m, int d, int y) {
    SetMonth(m);
    SetDay(d);
    SetYear(y);
}
```
Using Default Parameters

• the following are all valid declarations:

  Date graduation(5,18,2015);
  Date today;
  Date halloween(10,31);
  Date july(4);
Using Default Parameters

• the following are all valid declarations:

```c
Date graduation(5,18,2015);
Date today;
Date halloween(10,31);
Date july(4);
```

// graduation: 5/18/2015
// today: 10/15/2014
// halloween: 10/31/2014
// july: 4/15/2014
Using Default Parameters

• the following are all valid declarations:

```java
Date graduation(5,19,2014);
Date today;
Date halloween;
Date july(4);

// graduation: 5/19/2014
// today: 10/15/2014
// halloween: 10/31/2014
// july: 4/15/2014
```

NOTE: when you call a constructor with no arguments, you do not give it empty parentheses
Default Constructors

• a default constructor is provided by compiler
  – will handle declarations of Date instances

• this is how we created Date objects in the slides before we declared and defined our constructor
Default Constructors

• **but**, if you create **any** other constructor, the compiler doesn’t provide a default constructor

• so if you create a constructor, make a default constructor too, even if its body is just empty

```cpp
Date::Date ()
{
    /* empty */
}
```
Function Overloading

• functions in C++ are uniquely identified by both their names and their parameters
  – but NOT their return type!

• we can overload any kind of function
  – we can even use default values, like with constructors
Overloading Example

```c
void PrintMessage (void) {
    cout << "Hello World!" << endl;
}

void PrintMessage (string msg) {
    cout << msg << endl;
}
```
Outline

• Procedural Programming vs OOP
• Classes
  – Example: Morphing from Struct
  – Basics
  – Access
  – Constructors
  – Overloading
• Livecoding
Example: Rectangle Class

• width and height member variables
• accessors and mutators
• functions for IsSquare(), CalcArea(), CalcPerim(), and PrintRectInfo()

• what happens when we give a constructor that uses default parameters and calls mutators invalid arguments?
Parts of a Class (so far)

• class name
• member variables
  • constructors
    • default parameters
  • accessors
  • mutators
  • auxiliary (private and public)