

**CS100**

# **Introduction to Programming**

**Lecture 11. Object-Oriented Programming:  
Encapsulation**

# Learning objectives

- Understand the difference between
  - Procedural programming
  - Object-Oriented programming
- Understanding the role of a class in C++
- Access specifiers, Constructors & Overloading
- Code organization and compilation in C++

# Outline

- Procedural Programming vs OOP
- Classes
  - Example: Morphing from Struct
  - Basics
  - Access
  - Constructors
  - Overloading
- Code organization
- Compilation in C++

# Procedural Programming

- In C, 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
  - program calls the procedures in sequence
- Example:
  - `printf(<character array>, <parameters>)`

# Object-Oriented Programming

- now that we start 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

- With lots of data and tasks → unmaintainable
- Idea of OOP:
  - Concept of “interacting objects”
  - Data and procedures specific for an object are “packed away” into neat, self-contained boxes
  - Permits to think of objects more abstractly and focus on their interactions
- $C + OOP = C++!$

<https://www.youtube.com/watch?v=JBjinqGOBP8>

**Bjarne Stroustrup**

Inventor of C++



# 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 grouped together (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 (i.e. sub-classes)
- 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
  - ability to decide at runtime what will be done
- 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
- Code organization
- Compilation in C++

# Example Struct: Date

```
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?

```
void PrintDate (DATE day) ;
```

- if we want to change the year of a date, what should our function prototype be?

```
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

```
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
- Code organization
- Compilation in C++

# Functions in Classes

- unlike structs, classes have *member functions* along with their member variables
  - Note: `struct` refers to a C-style `struct`. In C++, there is (almost) no difference between `class` and `struct`
- member functions go inside the class declaration
- member functions are called on an object of that class type

```
iStream.open("file.txt");
```

object                      method

# Example: OutputMonth() Function

- let's add a function to the class that will print out the name of the month

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

# Example: OutputMonth()

- let's add a function to the class that will print out the name of the month

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

function  
prototype



# Example: OutputMonth()

```
void OutputMonth () ;
```

- nothing is passed in to the function – why?
- 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

```
void Date::OutputMonth () {
```

```
}
```

# OutputMonth() Definition

```
void Date::OutputMonth () {
```

specify class name;  
more than one class  
can have a function  
with the same name

```
}
```

# OutputMonth() Definition

```
void Date::OutputMonth () {
```

this double colon is called the *scope resolution operator*, and associates the *member function* `OutputMonth()` with the class `Date`

```
}
```

# OutputMonth() Definition

```
void Date::OutputMonth() {  
    switch (m_month) {  
        case 1: printf("January"); break;  
        case 2: printf("February"); break;  
        case 3: printf("March"); break;  
        /* etc */  
        default:  
            printf("Error in Date::OutputMonth\n");  
    }  
}
```

# OutputMonth() Definition

```
void Date::OutputMonth () {  
    switch (m_month) {  
        case 1: pr  
        case 2: pr  
        case 3: pr  
        /* etc */  
        default:  
            printf("Error in Date::OutputMonth\n");  
    }  
}
```

we can directly access `m_month` because it is a *member variable* of the `Date` class, to which `OutputMonth()` belongs

# Print Functions

- is the following valid code?  
`printf (today.OutputMonth () ) ;`
- no, because `OutputMonth ()` returns nothing for `printf` to print
  - if the function returned a string, this would be valid code

# 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

```
Date today;
```

```
printf("Please enter dates as DD MM YYYY:\n");
```

```
printf("Please enter today's date: ");
```

```
scanf("%d %d %d", &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

```
Date today;
```

```
printf("Please enter dates as DD MM YYYY:\n");
```

```
printf("P on the Date object today
```

```
scanf("%d
```

```
&today
```

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`

```
printf("Today's date is ");
```

```
today.OutputMonth();
```

```
printf("%d, %d\n", today.m_day, today.m_year);
```

# Outline

- Procedural Programming vs OOP
- **Classes**
  - Example: Morphing from Struct
  - Basics
  - **Access**
  - Constructors
  - Overloading
- Code organization
- Compilation in C++

# 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;  
};
```

# 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

- Many classifications.
- Example:
  - 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:  
    **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 ;  
};
```



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

# Outline

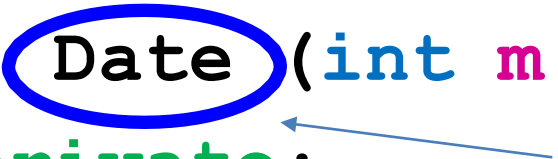
- Procedural Programming vs OOP
- **Classes**
  - Example: Morphing from Struct
  - Basics
  - Access
  - **Constructors**
  - Overloading
- Code organization
- Compilation in C++

# 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 ;  
};
```



exact same  
name as  
the class

# Date Class Constructors

```
class Date {  
public:  
    void OutputMonth() ;  
    Date (int m, int d, int y) ;  
    ~Date() :  
        _month;  
        _day;  
        int m_year;  
};
```

No return  
type, not  
even void

# Constructor Definition

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

}
}
```

# Constructor Definition

```
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)
{

    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; }
}
```

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**
- Code organization
- Compilation in C++

# 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

```
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

```
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

```
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:

```
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:

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

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

```
Date::Date () ;
```

- would the following be a valid constructor?

```
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

```
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) ;
```

```
// 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:

```
Date graduation(5,19,2014);
```

```
Date today;
```

```
Date halloween;
```

```
Date july(4);
```

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

```
// graduation: 5/19/2014
```

```
// today: 10/15/2014
```

```
// halloween: 10/31/2014
```

```
// july: 4/15/2014
```

# 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 own 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

```
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

```
void PrintMessage (void) {  
    printf("Hello World!");  
}
```

```
void PrintMessage (string msg) {  
    printf(msg.c_str());  
}
```

# Outline

- Procedural Programming vs OOP
- Classes
  - Example: Morphing from Struct
  - Basics
  - Access
  - Constructors
  - Overloading
- Code organization
- Compilation in C++

# Code organization

- Header: Contains class declarations with constructors, member function, and variables
- Source-file: Contains implementations
- Abstraction into interacting objects:
  - One Header & Source-file per object!
  - Gain understanding of program concept (i.e. objects and their interactions) by looking at header files only!

# Code organization

- What about the following?
  - We have classes **Date**, **Name**, and **Location**
  - We have a class **Birthday** that includes
    - **Date** and **Name**
  - We have a class **Meeting** that includes
    - **Date** and **Location**
  - We have a class **Calendar** that includes
    - **Birthday** and **Meeting**
- Recursive resolving of `#include` will lead to double declaration of **Date**!

# Code organization

- Include guards ensure unique declaration!

```
#ifndef DATE_HPP_  
#define DATE_HPP_
```

```
class Date {  
    ...  
};
```

```
#endif
```

# Outline

- Procedural Programming vs OOP
- Classes
  - Example: Morphing from Struct
  - Basics
  - Access
  - Constructors
  - Overloading
- Code organization
- **Compilation in C++**



# Compilation in C++

- instead of **gcc** use **g++**
- you can still use the same flags:
  - Wall** for all warnings
  - c** for denoting separate compilation
  - o** for naming an executable
  - g** for allowing use of a debugger
    - and any other flags you used with gcc

# Compilation in C++

- Compiling multiple files:
  - `g++ main.cpp Class1.cpp Class2.cpp -o main`