

CS100

Introduction to Programming

Lecture 16. 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

<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

Outline

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

Functions in Classes

- unlike C 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

```
myVector.push_back(newVal);
```

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() const;  
};
```

function
prototype

Example: OutputMonth()

```
class Date {  
    //...  
    void OutputMonth() const;  
};
```

- nothing is passed in to the function – why?
- because `m_month` is a *member variable* of the Date class
 - just like `OutputMonth()` is a *member function*
- a const member function - why?
- because it only needs *access* to `m_month`

OutputMonth() Definition

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

```
}
```

OutputMonth() Definition

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

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

```
}
```

OutputMonth() Definition

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

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

```
}
```

OutputMonth() Definition

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

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;
```

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

```
cout<<"Please enter today's date: "<<endl;
```

```
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

```
Date today;
```

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

```
cout<<"Please enter today's date: "<<endl;
```

```
cin>>today.m_day >> today.m_month >> today.m_year;
```

```
cout<<"Today's date is "<<endl;
```

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

```
cout<<today.m_day<<" , "<< today.m_year<<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`

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

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.
- In a **class**, if not specified, members default to private
 - should specify anyway – good coding practices!
- **struct** and **class** differ: Members in a **struct** default to public.

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() const;  
    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() const;  
int GetDay() const;  
int GetYear() const;
```

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() const; → public`
`void IncrementDate(); → private`

Access Specifiers for Date Class

```
class Date {  
public:  
    void OutputMonth() const;  
    int  GetMonth() const;  
    int  GetDay() const;  
    int  GetYear() const;  
    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

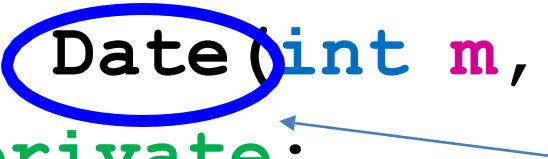
- Procedural Programming vs OOP
- **Classes**
 - Example: Morphing from Struct
 - Basics
 - Access
 - **Constructors**
 - Overloading
- Code organization
- Compilation in C++

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

No return-type declaration.
(It is not void.)

```
    int m_day;  
    int m_year;  
};
```

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) {}
```

- Better: Use *constructor initializer list*.
- Members are *initialized* in the order in which they appear in class definition.
 - Warning if it does not match the order in the initializer list.
 - Use the value of an uninitialized member variable is *undefined behavior*!
- Also initializes const member variables.
 - A const member variable cannot be *assigned* a value in the constructor body.

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

Overloaded constructors

- Define multiple constructors (different ways of constructing)

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

```
Date::Date(int yyymmdd)
    : m_month(yyymmdd / 100 % 100),
      m_day(yyymmdd % 100),
      m_year(yyymmdd / 10000) { }
```

Overloaded Constructors

- Suppose we have the following constructors:

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

```
Date::Date(int m, int d)
    : m_month(m), m_day(d), m_year(1)
{ }
```

```
Date::Date()
    : m_month(1), m_day(1), m_year(1)
{ }
```

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)
    : m_month(m) , m_day(d) , m_year(y) {}
```

or we can combine them together:

```
class Date {
public:
    Date(int m, int d, int y)
        : m_month(m) , m_day(d) , m_year(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 (**if you really need it**), and define it in a *reasonable* way.
 - If the class does not have a reasonable way of default-construction, do not define it.
 - *More Effective C++* Item 4: Avoid gratuitous default constructors.

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`