

CS-202

C++ Classes (Introduction)

C. Papachristos

Autonomous Robots Lab
University of Nevada, Reno



Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	Project DEADLINE	NEW Project	

Your 2nd Project Deadline is this Wednesday 2/7.

- PASS Sessions held Monday-Tuesday
 - RL Session held Wednesday
- } get all the help you may need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
 - Past that, NO Project accepted. Better send what you have in time!

Today's Topics

C++ Classes

- Definitions
- Declaration, Implementation
- Members, Methods
- Usage, Coding Standards

Classes as Abstract Data Types

Protection Mechanisms

Programming Abstraction

All programming languages provide some form of Abstraction.

- Also called “Information Hiding”.
- Separates code use from code implementation.

In *Procedural* Programming:

- Data Abstraction: Data Structures. `struct somethingComplex{ ... };`
- Control Abstraction: Functions. `void makeItHappen(...);`

In *Object-Oriented* Programming

- Data and Control Abstraction: Using Classes

Programming Abstraction

All programming languages provide some form of Abstraction.

Not to be confused with Abstract Types:

- A programming language-related implementation.
- Given a type system, an Abstract Type is one that cannot be *instantiated* directly (vs a Concrete Type).

`<abstract_type> Vehicle ;` `<concrete_type> Vehicle : Car ;`



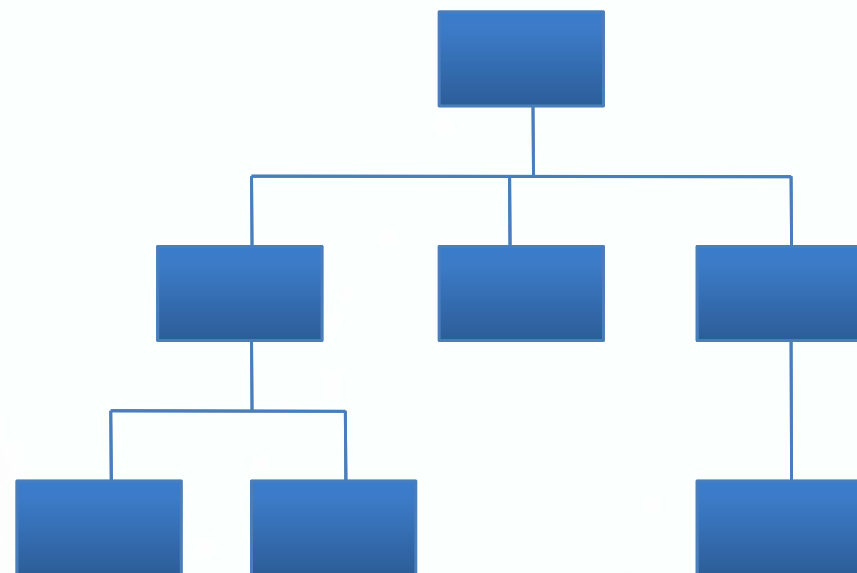
Remember: Procedural *vs* Object-Oriented

Procedural

Procedural Decomposition:

Divides the problem into more easily handled subtasks, until the functional modules (subproblems) can be coded.

Focus on: Processes.



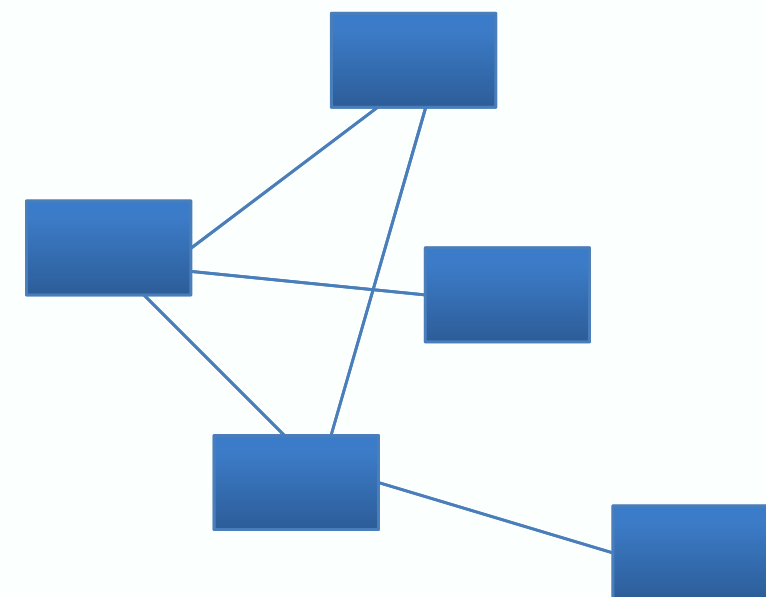
A hierarchy
of functions

Object-Oriented (OO)

Object-Oriented Design:

Identifies various objects composed of data and operations, that can be used together to solve the problem.

Focus on: Data Objects.



A collection
of Objects

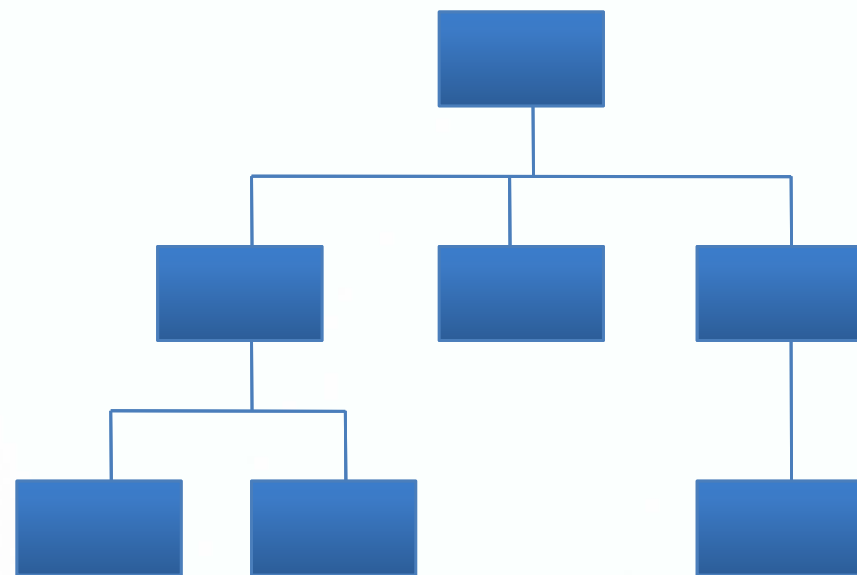
Remember: Procedural *vs* Object-Oriented

Procedural

Focused on the question: “What should the program do next?” Structure program by:

- Splitting into sets of tasks and subtasks.
- Make functions for tasks.

- Data and operations are not bound to each other



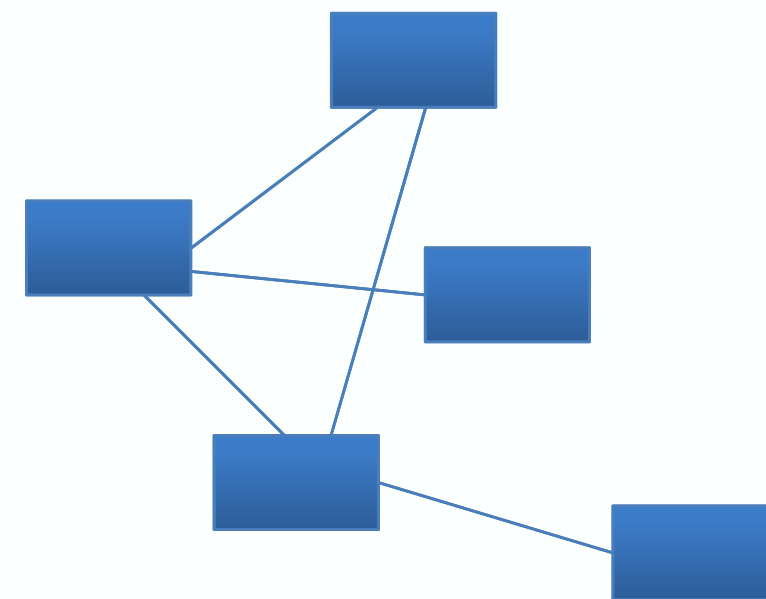
A hierarchy
of functions

Object-Oriented (OO)

Package-up self-sufficient modular pieces of code. Pack away details into boxes (objects) keep them in mind in their abstract form.

- “The world is made up of interacting objects”.

- Data and operations are bound to each other.



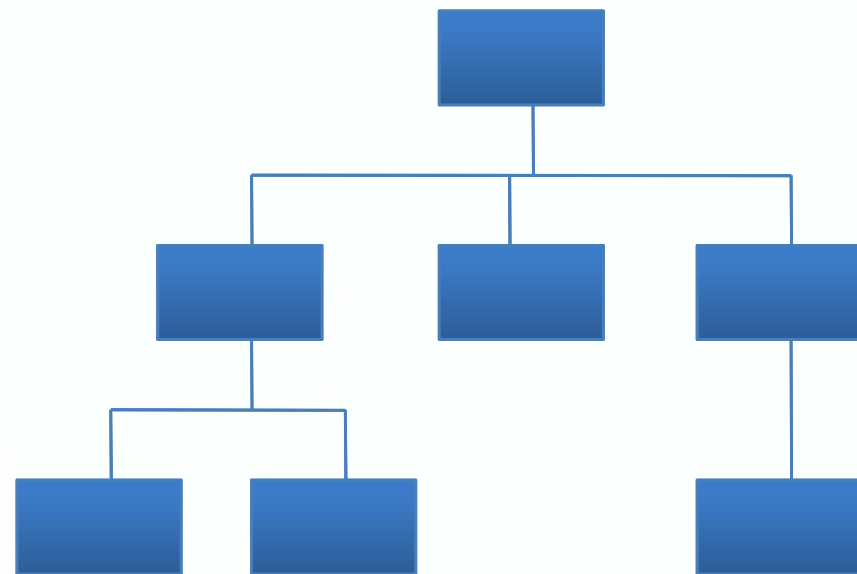
A collection
of Objects

Remember: Procedural vs Object-Oriented

Procedural

“What should the program do next?”

- Calculate the area of a circle **given** the specified radius.
- Sort this class list **given** an array of students.
- Calculate the car's expected **mileage** given its gas and road conditions.

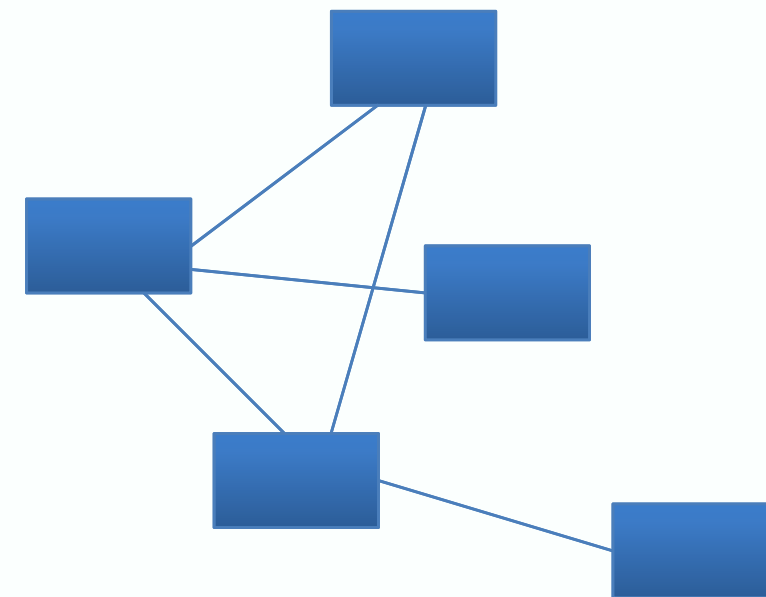


A hierarchy
of functions

Object-Oriented (OO)

Self-sufficient, modular, interacting pieces of code.

- **Circle**, you know your radius, what is your area?
- **Class list**, sort your students.
- **Car**, when will you run out of gas on this trip?



A collection
of Objects

Object-Oriented Programming

Principles

Information Hiding:

- Details of how operations work are not known to the user of the Class.

Data Abstraction:

- Details of how data is manipulated within “Abstract Data Type” / Class are not known to the user.

Encapsulation:

- Bring together data and operations, but keep details hidden.

Object-Oriented Programming

Classes

According to the dictionary:

- “A kind or category.”
- “A set, collection, group, or configuration containing members regarded as having certain attributes or traits in common.”

According to OOP principles:

- A group of objects with similar properties, common behavior, common relationships with other objects, and common semantics.
- We use *Classes* for Abstraction purposes.

Blueprints

Classes are “blueprints” for creating Objects.

- A ***Dog*** Class to create ***dog*** Objects.
- A ***Car*** Class to create ***car*** Objects.
- A ***Shoe*** Class to create ***shoe*** Objects.

The blueprint defines:

- The Class’s state/attributes as *class member variables*.
- The Class’s behaviors as *class methods*.

Objects

Variables of Class types may be created just like variables of built-in types:

- Each instance of a class is called an Object of that Class type.
- Using a set of *Car* blueprints we can create a *car* Object.

We can create as many instances of a Class as needed:

- Just like a regular data type, *int*, *float*, etc.
- There is more than one *dog*, *car*, *shoe* (and might differ a lot)!

The challenge is to define Classes and create Objects that satisfy the problem:

- Do we need a *Car* class?

Class Interface

The requests you can make of an Object are determined by its interface.

Do we need to know?

- *How* the **car** *manufacturing* chain works in order to buy one?
- *How* the **car** operates *internally* in order to drive one?

All we need to know is:

- How the **car** dealership works with financing.
How to get one?
- How the **car** pedals, signals, switches, and steering wheel work.
How to operate one?

Class Interface

The requests you can make of an Object are determined by its interface.

- *How to get one?*
- *How to operate one?*

<i>Car Class</i>	Type
Dealership price/scheme	Interface
Operate steering wheel	
Operate gas pedal	
Operate brake pedal	
Operate clutch	
Operate transmission	
Switch lights	
...	

Class Implementation

What actually lies inside the Class. It is the:

- Code,
- Hidden Data,

that satisfy requests made to it (and/or its Objects).

Every request made of an Object must have associated *Method* (i.e. Function) that will be called.

- When dealing with OO content, we say that the user is sending a message to the object, which responds to the message by executing the appropriate code.
- “The world is made up of interacting objects”.

Classes

Class Declaration

```
class Car  
{  
    public:  
    bool AddGas(float gallons);  
    float GetMileage();  
    // other operations  
    private:  
    float m_currGallons;  
    float m_currMileage;  
    // other data  
};
```

Class (Type) Name

Classes

Class Declaration

```
class Car
{
    public:
    bool AddGas(float gallons);
    float GetMileage();
    // other operations
    private:
    float m_currGallons;
    float m_currMileage;
    // other data
};
```

Class (Type) Name

Protection Mechanism

Protection Mechanism

Classes

Class Declaration

```
class Car
{
    public:
    bool AddGas(float gallons);
    float GetMileage();
    // other operations

    private:
    float m_currGallons;
    float m_currMileage;
    // other data
};
```

Class (Type) Name

Protection Mechanism

Protection Mechanism

Data

Classes

Class Declaration

```
class Car
{
    public:
    bool AddGas(float gallons);
    float GetMileage();
    // other operations

    private:
    float m_currGallons;
    float m_currMileage;
    // other data
};
```

Class (Type) Name

Protection Mechanism

Operations

Protection Mechanism

Data

Class Conventions

Standards for coding with Classes:

`class Car`

This is already *Italicized*!

Integrated Development Environments can sometimes save the day with their smart features:

- Real-time search for Declaration.
- Auto-completion, Function alternatives.

But:

- Learn to adopt a set of conventions (not rules), same as with every other language.

Class Conventions

Class names:

- Always begin with capital letter.
- Use **mixedCase** for phrases.
- General word for Class (Type) of Objects.

Examples: **Car**, **Boat**, **DVD**, **List**, **Customer**, **BoxOfDVDs**, ...

```
class Car  
{  
    ...  
};
```

Class Conventions

Class data (member variables):

- Always begin names with **m_** (stands for “member”).
Examples: `float m_fuel`, `char* m_title`, ...

```
class Car
{
    ...
    float m_currGallons;
    float m_currMileage;
};
```

Class Conventions

Class operations/methods:

- Begin with capital letter.

Examples: `AddGas()`, `Accelerate()`, `ModifyTitle()`, `RemoveDVD()`, ...

```
class Car
{
    ...
    bool AddGas(float gallons);
    float GetMileage();
};
```

Encapsulation

Main principle in Object-Oriented Design / Programming.

- A form of “Information Hiding” and Abstraction.

How:

- **Data** and **Functions that act on *that* data** are located in the same place.
- Encapsulated inside the Class.

Goal:

- Separate *Interface* from *Implementation*.
Someone can still use the code without any knowledge of how it works!

Encapsulation

Classes encapsulate both Data and Functions.

- Class definitions must contain both!

Member Variables are the Data of a Class.

- Its attributes, characteristics, an Object's state.
(e.g. **breed** of *Dog*, **size** of *Shoe*, **make** of *Car* ...)

Class Methods are used to act on that Data.

- (e.g., **Play**() with *Dog*, **Inspect**() a *Car*, ...)

BankAccount

Member Vars:

m_AccountNr

m_OwnerName

m_Balance

Class Methods:

DepositMoney()

WithdrawMoney()

CheckBalance()

Class Components

Member variables:

- What data must be stored?

Class Methods/Member Functions:

- How does the user need to interact with the stored data?

Constructor(s):

- How do you build an instance?

Destructor:

- How do you clean up an after an instance?

Class, by-Example

// Represents a Day of the Year

```
class DayOfYear
```

```
{
```

```
    public:
```

```
        void Output();
```

```
        int m_month;
```

```
        int m_day;
```

```
};
```

// Output method - displays a DayOfYear

```
void DayOfYear::Output( )
```

```
{
```

```
    cout << m_month << "/" << m_day;
```

```
}
```

Class Name

Access Specifier

Method(s)

Data

Classes

Class, by-Example

// Represents a Day of the Year

```
class DayOfYear
```

```
{
```

```
    public:
```

```
        void Output();
```

```
        int m_month;
```

```
        int m_day;
```

```
};
```

// Output method - displays a DayOfYear

```
void DayOfYear::Output( )
```

```
{
```

```
    cout << m_month << "/" << m_day;
```

```
}
```

Class Name

Access Specifier

Method(s)

Data

Method Prototype *inside*
Class Declaration

Method Implementation *outside* of
Class Declaration

Class Method Implementation

The Method Implementation:

```
// Output method - displays a DayOfYear  
return type Method Name & Parameters List  
void DayOfYear::Output( )  
{  
    cout << m_month << "/" << m_day;  
}
```

Class Method Implementation

The Method Implementation:

// Output method - displays a DayOfYear

Scope Resolution Operator (::)

Indicates which Class Method this definition implements.

Simpler: Which Class is it from?

Class Name



```
void DayOfYear::Output ( )  
{  
    cout << m_month << "/" << m_day;  
}
```

Class Method Implementation

The Method Implementation:

// Output method - displays a DayOfYear

Scope Resolution Operator (::)

Indicates which Class Method this definition implements.

Simpler: Which Class is it from?

Class Name

```
void DayOfYear::Output ( )  
{  
    cout << m_month << "/" << m_day;  
}
```

Method Body
Access to
Class Member Variables

Class Separation into Files

```
// Represents a Day of the Year
class DayOfYear
{
    public:
        void Output();
        int m_month;
        int m_day;
};
```

```
// Output method - displays a DayOfYear
void DayOfYear::Output( )
{
    cout << m_month << "/" << m_day;
}
```

Class Declaration:

- Goes into Class header file.

<ClassName.h>

<DayOfYear.h>

Class Definition:

- Goes into Class source file.

<ClassName.cpp>

<DayOfYear.cpp>

Classes

Class Usage

The Method Implementation:

```
// Inside a main() somewhere  
DayOfYear july4th;
```

Constructor

Object
Name

```
july4th.m_month = 7;  
july4th.m_day = 4;  
july4th.Output();
```

Class Usage

The Method Implementation:

```
// Inside a main() somewhere
```

```
DayOfYear July4th;
```

Constructor

Dot Operator (.) – Member-of

Indicates which Object this Class Member references.

Simpler: The Member-of which Object?

Object
Name

```
July4th.m_month = 7;  
July4th.m_day = 4;  
July4th.Output();
```

Class Usage

The Method Implementation:

```
// Inside a main() somewhere
```

```
DayOfYear July4th;
```

Constructor

Dot Operator (.) – Member-of

Indicates which Object this Class Member references.

Simpler: The Member-of which Object?

Object
Name

```
July4th.m_month = 7;  
July4th.m_day = 4;  
July4th.Output();
```

Class Member Variables
&
Class Methods

Class Usage

The Method Implementation:

```
// Inside a main() somewhere
```

```
DayOfYear july4th;
```

```
DayOfYear* july4th_Pt = &july4th;
```

Pointer to Class Type

Object
Pointer

```
july4th_Pt->m_month = 7;  
july4th_Pt->m_day = 4;  
july4th_Pt->Output();
```


Class Usage

The Method Implementation:

```
// Inside a main() somewhere
```

```
DayOfYear july4th;
```

```
DayOfYear* july4th_Pt = &july4th;
```

Pointer to Class Type

Arrow Operator (->) – Member-access

Class Pointer Dereference Operator (The C++ standard just calls it “arrow” (§5.2.5)).

Simpler: “Works out” similarly to Member-of (.).

Object
Pointer

```
july4th_Pt->m_month = 7;  
july4th_Pt->m_day = 4;  
july4th_Pt->Output();
```

Class, by-Example (one more)

```
1 //Program to demonstrate a very simple example of a class.
2 //A better version of the class DayOfYear will be given in Display 6.4.
3 #include <iostream>
4 using namespace std;
```

*Normally, member variables are **private** and not **public**, as in this example. This is discussed a bit later in this chapter.*

```
5 class DayOfYear
6 {
7 public:
8     void output( );
9     int month;
10    int day;
11 };
```

Member function declaration

```
12 int main( )
13 {
14     DayOfYear today, birthday;
15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
17     cin >> today.month;
18     cout << "Enter the day of the month: ";
19     cin >> today.day;
20     cout << "Enter your birthday:\n";
21     cout << "Enter month as a number: ";
22     cin >> birthday.month;
23     cout << "Enter the day of the month: ";
24     cin >> birthday.day;
```

```
25     cout << "Today's date is ";
26     today.output( );
27     cout << endl;
28     cout << "Your birthday is ";
29     birthday.output( );
30     cout << endl;
31     if (today.month == birthday.month && today.day == birthday.day)
32         cout << "Happy Birthday!\n";
33     else
34         cout << "Happy Unbirthday!\n";
35     return 0;
36 }
```

Calls to the member function output

Class, by-Example (one more)

```
37 //Uses iostream:
38 void DayOfYear::output( )
39 {
40     switch (month)
41     {
42         case 1:
43             cout << "January "; break;
44         case 2:
45             cout << "February "; break;
46         case 3:
47             cout << "March "; break;
48         case 4:
49             cout << "April "; break;
50         case 5:
51             cout << "May "; break;
52         case 6:
53             cout << "June "; break;
54         case 7:
55             cout << "July "; break;
56         case 8:
57             cout << "August "; break;
58         case 9:
59             cout << "September "; break;
60         case 10:
61             cout << "October "; break;
62         case 11:
63             cout << "November "; break;
64         case 12:
65             cout << "December "; break;
66         default:
67             cout << "Error in DayOfYear::output. Contact software vendor.";
68     }
69     cout << day;
71 }
```


Class, by-Example (one more)

```
1 //Program to demonstrate a very simple example of a class.
2 //A better version of the class DayOfYear will be given in Display 6.4.
3 #include <iostream>
4 using namespace std;
```

*Normally, member variables are **private** and not **public**, as in this example. This is discussed a bit later in this chapter.*

```
5 class DayOfYear
6 {
7 public:
8     void output( );
9     int month;
10    int day;
11};
```

Member function declaration

Note:
Properly, this is placed in
<DayOfYear.h>

```
12 int main( )
13 {
14     DayOfYear today, birthday;
15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
17     cin >> today.month;
18     cout << "Enter the day of the month: ";
19     cin >> today.day;
20     cout << "Enter your birthday:\n";
21     cout << "Enter month as a number: ";
22     cin >> birthday.month;
23     cout << "Enter the day of the month: ";
24     cin >> birthday.day;
```

```
25     cout << "Today's date is ";
26     today.output( );
27     cout << endl;
28     cout << "Your birthday is ";
29     birthday.output( );
30     cout << endl;
31     if (today.month == birthday.month && today.day == birthday.day)
32         cout << "Happy Birthday!\n";
33     else
34         cout << "Happy Unbirthday!\n";
35     return 0;
36 }
```

Calls to the member function output

Class, by-Example (one more)

```
37 //Uses iostream:
38 void DayOfYear::output( )
39 {
40     switch (month)
41     {
42         case 1:
43             cout << "January "; break;
44         case 2:
45             cout << "February "; break;
46         case 3:
47             cout << "March "; break;
48         case 4:
49             cout << "April "; break;
50         case 5:
51             cout << "May "; break;
52         case 6:
53             cout << "June "; break;
54         case 7:
55             cout << "July "; break;
56         case 8:
57             cout << "August "; break;
58         case 9:
59             cout << "September "; break;
```

```
60         case 10:
61             cout << "October "; break;
62         case 11:
63             cout << "November "; break;
64         case 12:
65             cout << "December "; break;
66         default:
67             cout << "Error in DayOfYear::output. Contact software vendor.";
68     }
69
70     cout << day;
71 }
```

Note:

Properly, this is placed in
<DayOfYear.cpp>

A Class' Place

A Class is full-fledged *Type*!

- Just like data types `int`, `double`, etc.

Hence, we can have Variables *of a Class Type*:

- We simply call them “Objects”.

Can have Function Parameters of a Class Type.

- Pass-by-Value.
- Pass-by-Reference.
- Pass-by-Address.

Pass-by-Value

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Value.

```
DayOfYear july4th;  
july4th.m_month = 7; july4th.m_day = 4;  
printNextDay(july4th);  
void printNextDay(DayOfYear date) {  
    date.m_day++;  
    if (date.m_day ... && date.m_month ...) {  
        date.m_day = ...;  
        date.m_month = ...;  
    }  
    date.Output();  
}
```

```
class DayOfYear{  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Pass-by-Value

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Value.

```
DayOfYear july4th;  
july4th.m_month = 7; july4th.m_day = 4;  
printNextDay(july4th);  
void printNextDay(DayOfYear date) {  
    date.m_day++;  
    if (date.m_day ... && date.m_month ...) {  
        date.m_day = ...;  
        date.m_month = ...;  
    }  
    date.Output();  
}
```

```
class DayOfYear{  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Note:
Will work with Local Object Copy !

Pass-by-Reference

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Reference.

```
DayOfYear july4th;  
july4th.m_month = 7; july4th.m_day = 4;  
shiftNextDay(july4th);  
july4th.Output();  
  
void shiftNextDay(DayOfYear& date) {  
    date.m_day++;  
    if (date.m_day ... && date.m_month ...) {  
        date.m_day = ...;  
        date.m_month = ...;  
    }  
}
```

```
class DayOfYear{  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Pass-by-Reference

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Reference.

```
DayOfYear july4th;  
july4th.m_month = 7; july4th.m_day = 4;  
shiftNextDay(july4th);  
july4th.Output();  
  
void shiftNextDay(DayOfYear& date) {  
    date.m_day++;  
    if (date.m_day ... && date.m_month ...) {  
        date.m_day = ...;  
        date.m_month = ...;  
    }  
}
```

```
class DayOfYear {  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Note:

Will modify Object Data !

Pass-by-Address

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Address.

```
DayOfYear july4th;  
DayOfYear* july4th_Pt = &july4th;  
shiftNextDay_Pt(july4th_Pt);  
july4th.Output();  
  
void shiftNextDay_Pt(DayOfYear* date_p) {  
    date_p->m_day++;  
    if(date_p->m_day ... && date_p->m_month ...) {  
        date_p->m_day = ...;  
        date_p->m_month = ...;  
    }  
}
```

```
class DayOfYear{  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Pass-by-Address

Hence, we can also have Function Parameters of a Class derivatives:

- Function Parameter by-Address.

```
DayOfYear july4th;  
DayOfYear* july4th_Pt = &july4th;  
shiftNextDay_Pt(july4th_Pt);  
july4th.Output();
```

```
void shiftNextDay_Pt(DayOfYear* date_p) {  
    date_p->m_day++;  
    if(date_p->m_day ... && date_p->m_month ...) {  
        date_p->m_day = ...;  
        date_p->m_month = ...;  
    }  
}
```

```
class DayOfYear{  
    public:  
        void Output();  
        int m_month;  
        int m_day;  
};
```

Note:

Will modify Object Data !

Abstract Data Types

The concept of “Programming Abstraction” :

- *Programmers* don’t (need to) know the details!

Abbreviated “ADT” :

- An ADT is a collection of data values together with set of basic operations defined for the values, ADTs are often language-independent.
- In C++ we ADTs are implemented with Classes.

A C++ Class “defines” the ADT.

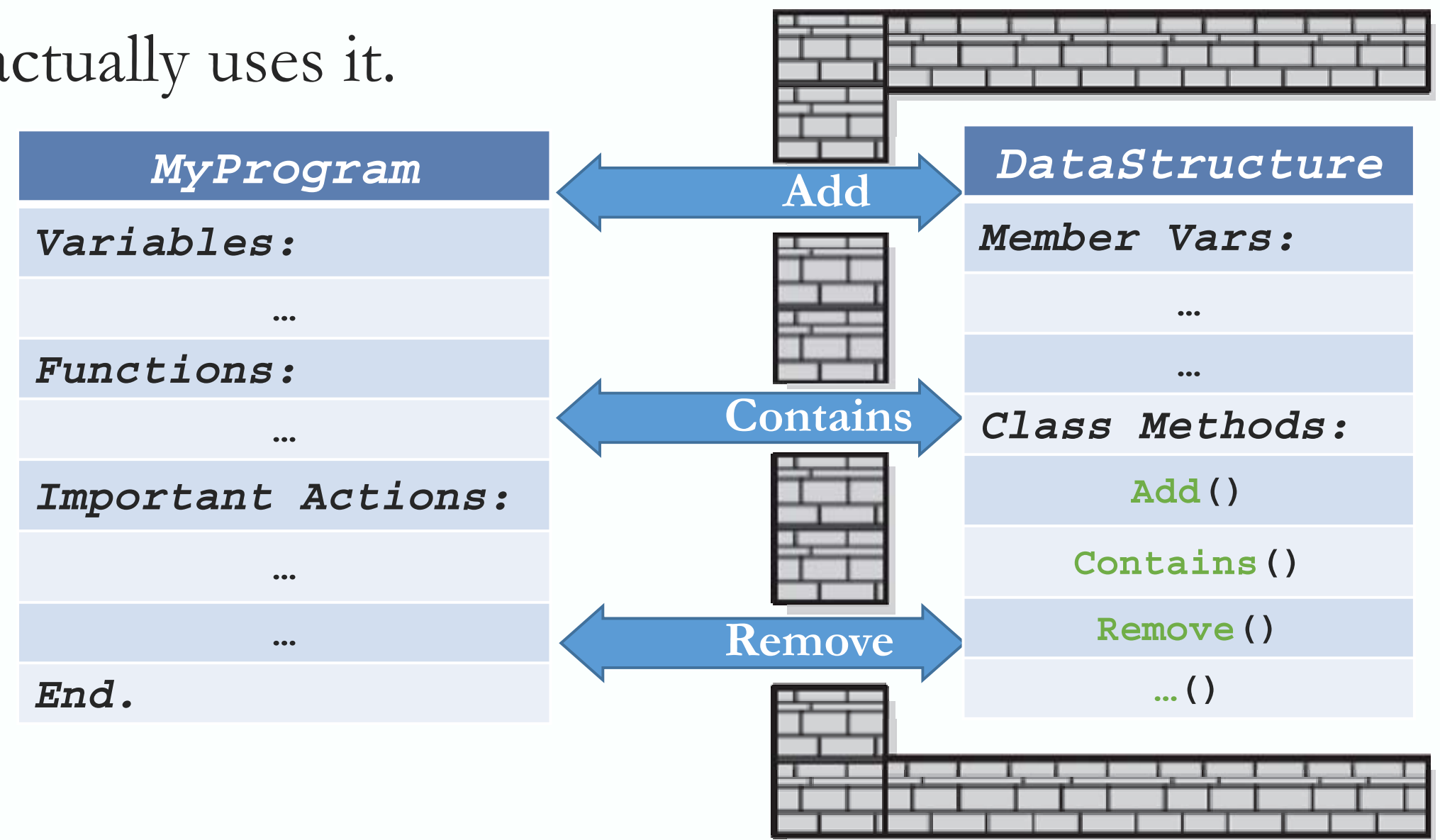
A Data Structure :

- An ADT implementation within a programming language.

Abstract Data Types

A wall of ADT operations isolates a Data Structure:

- from the program that actually uses it.



Coupling (more on Abstraction)

“Coupling” refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop-in a new implementation of it)

- Placing a new battery in a car *vs* a new engine.
- Adding a USB device *vs* a new video card to a laptop.

Object-Oriented Design seeks to reduce Coupling as much as possible by:

- Well-defined Interfaces to change (*write*) or access (*read*) the state of an Object.
- Enforcing those interfaces are adhered to (**private** *vs* **public**).
- Alternate implementations that may be more appropriate for different cases.

Encapsulation (*Reminder*)

Main principle in Object-Oriented Design / Programming.

- A form of “Information Hiding” and Abstraction.

How:

- **Data** and **Functions acting on *that* data** are placed in **same code unit**.
- Encapsulated inside the Class.

Goal:

- Separate *Interface* from *Implementation*.
Keep state separate from users via **private** Data, **public** Member Functions.
Someone can still use the code without any knowledge of how it works!

Encapsulation (a correlation to Classes)

Any data type includes:

- Data (range of Data).
- Operations (that can be performed on Data).

Example: The `int` data type.

Data: -2147483648 to 2147483647 (for 32-bit `int` – a.k.a. `int32_t`)

Operations: `+`, `-`, `*`, `/`, `%`, logical, etc.

The same holds with Classes:

- But Data are specified by user/programmer/you(!), and operations to be allowed on that Data (and their implementation) by you(!) as well.

Classes

Encapsulation (a correlation to Classes)

In one sense, it means “bringing together as one”:

- Declare & Define a Class.

```
class DayOfYear
{
    public:
        void Output();
        int m_month;
        int m_day;
};
```

```
void DayOfYear::Output() {
    cout << m_month << "/" << m_day;
}
```

```
DayOfYear july4th;
july4th.m_month = 7;
july4th.m_day = 4;
july4th.Output();
```

Encapsulation (a correlation to Classes)

In one sense, it means “bringing together as one”:

➤ Declare & Define a Class ➡ Get an Object.

```
class DayOfYear
{
    public:
        void Output();
        int m_month;
        int m_day;
};

void DayOfYear::Output() {
    cout << m_month << "/" << m_day;
}
```

```
DayOfYear july4th;
```

```
july4th.m_month = 7;
july4th.m_day = 4;
july4th.Output();
```

Encapsulation (a correlation to Classes)

In one sense, it means “bringing together as one”:

- Declare & Define a Class → Get an Object.
- The Object is “Encapsulation” of: a) Data values, b) Data operations.

```
class DayOfYear
{
    public:
        void Output();
        int m_month;
        int m_day;
};

void DayOfYear::Output() {
    cout << m_month << "/" << m_day;
}
```

```
DayOfYear july4th;

july4th.m_month = 7;
july4th.m_day = 4;
july4th.Output();
```


Encapsulation (a correlation to Classes)

Class Methods do not need to be passed information about that Class Object!

- Remember how the `DayOfYear::Output()` Method has no parameters?

Member Functions are called *on* a Class Object.

- They know everything about that object already. Why?
- It is the Object itself that applies the Data operations (Method).
It is the one that contains the Data, and its class contains the code.

Protection Mechanisms (continued)

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
class DayOfYear{  
    public:  
  
    void PrintDay() const;  
    void ShiftNextDay();  
  
    int m_month;  
    int m_day;  
};
```

```
void DayOfYear::PrintDay() const{  
    cout << m_month <<  
        "/" << m_day;  
}  
  
void DayOfYear::ShiftNextDay(){  
    m_day++;  
    if(m_day ... && m_month ...){  
        m_day = ...; m_month = ...;  
    }  
}
```

Protection Mechanisms

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
class DayOfYear{  
    public:
```

“Promises” to leave
Data untouched

```
    void PrintDay() const;  
    void ShiftNextDay();  
  
    int m_month;  
    int m_day;  
};
```

```
void DayOfYear::PrintDay() const{  
    cout << m_month <<  
        "/" << m_day;  
}
```

```
void DayOfYear::ShiftNextDay(){  
    m_day++;  
    if(m_day ... && m_month ...){  
        m_day = ...; m_month = ...;  
    }  
}
```

Protection Mechanisms

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
class DayOfYear{  
    public:
```

```
    void PrintDay() const;
```

```
    void ShiftNextDay();
```

```
    int m_month;  
    int m_day;
```

```
};
```

“Promises” to leave
Data untouched

Makes no such
“promise”.

```
void DayOfYear::PrintDay() const{  
    cout << m_month <<  
        "/" << m_day;  
}
```

```
void DayOfYear::ShiftNextDay(){  
    m_day++;  
    if(m_day ... && m_month ...){  
        m_day = ...; m_month = ...;  
    }  
}
```


Protection Mechanisms

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
class DayOfYear{  
    public:  
  
    void PrintDay() const;  
    void ShiftNextDay();  
  
    int m_month;  
    int m_day;  
};
```

```
void DayOfYear::PrintDay() const{  
    cout << m_month <<  
        "/" << m_day;  
}
```

Note (more on this later) :

In the body of a **cv**-qualified function, the **this** pointer is **cv**-qualified, e.g. in a **const** member function, only other **const** member functions may be called normally.

Protection Mechanisms

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
DayOfYear july4th;  
DayOfYear* july4th_Pt = &july4th;  
  
july4th.ShiftNextDay();  
july4th.PrintDay();  
  
july4th_Pt->ShiftNextDay();  
july4th_Pt->PrintDay();
```

```
class DayOfYear{  
    public:  
  
        void PrintDay() const;  
        void ShiftNextDay();  
  
        int m_month;  
        int m_day;  
};
```

Protection Mechanisms

The keyword **const** for Member Function(s):

- Member Functions have access to all Member Variables.
- Use **const** function signature to “promise” it won’t change Member Data.

```
DayOfYear july4th;  
DayOfYear* july4th_Pt = &july4th;
```

```
july4th.ShiftNextDay();
```

```
july4th.PrintDay();
```

```
july4th_Pt->ShiftNextDay();
```

```
july4th_Pt->PrintDay();
```

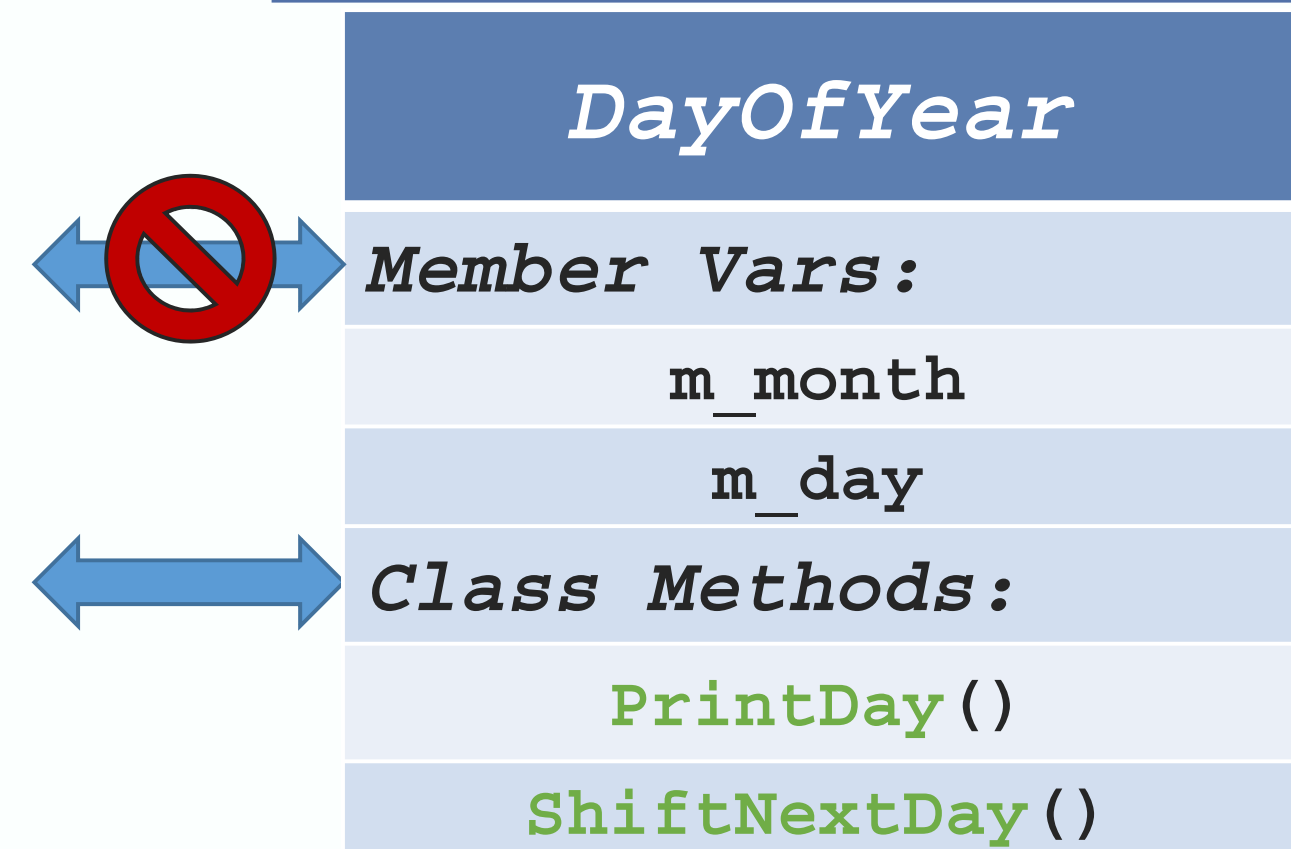
```
class DayOfYear{  
    public:  
        void PrintDay() const;  
        void ShiftNextDay();  
  
        int m_month;  
        int m_day;  
};
```

Protection Mechanisms

Access Specifiers:

```
class DayOfYear{  
    public:  
        void PrintDay() const;  
        void ShiftNextDay();  
    private:  
        int m_month;  
        int m_day;  
};
```

The CHANGE:
Data are now **private**.
Direct Object Interface to
Member Data is broken!



Protection Mechanisms

Access Specifiers:

```
class DayOfYear{  
    public:  
        void PrintDay() const;  
        void ShiftNextDay();  
    private:  
        int m_month;  
        int m_day;  
};
```

The CHANGE:
Data are now **private**.
Direct Object Interface to
Member Data is broken!

```
DayOfYear july4th;  
july4th.m_month = 7;  
july4th.m_day = 4;  
july4th.ShiftNextDay();  
july4th.Output();  
cout << july4th.m_day;
```

Impossible

Impossible

Protection Mechanisms

Access Specifiers style:

Can mix & match **public** & **private**:

- More typically place **public** first
- Allows easy viewing of portions that actually can be used by programmers using the Class.

private data is “hidden”, so irrelevant to users of Class.

- Outside of Class definition, cannot change (or access) **private** data.

Protection Mechanisms

Accessor & Mutator Functions:

Object needs to “do something” with its data !

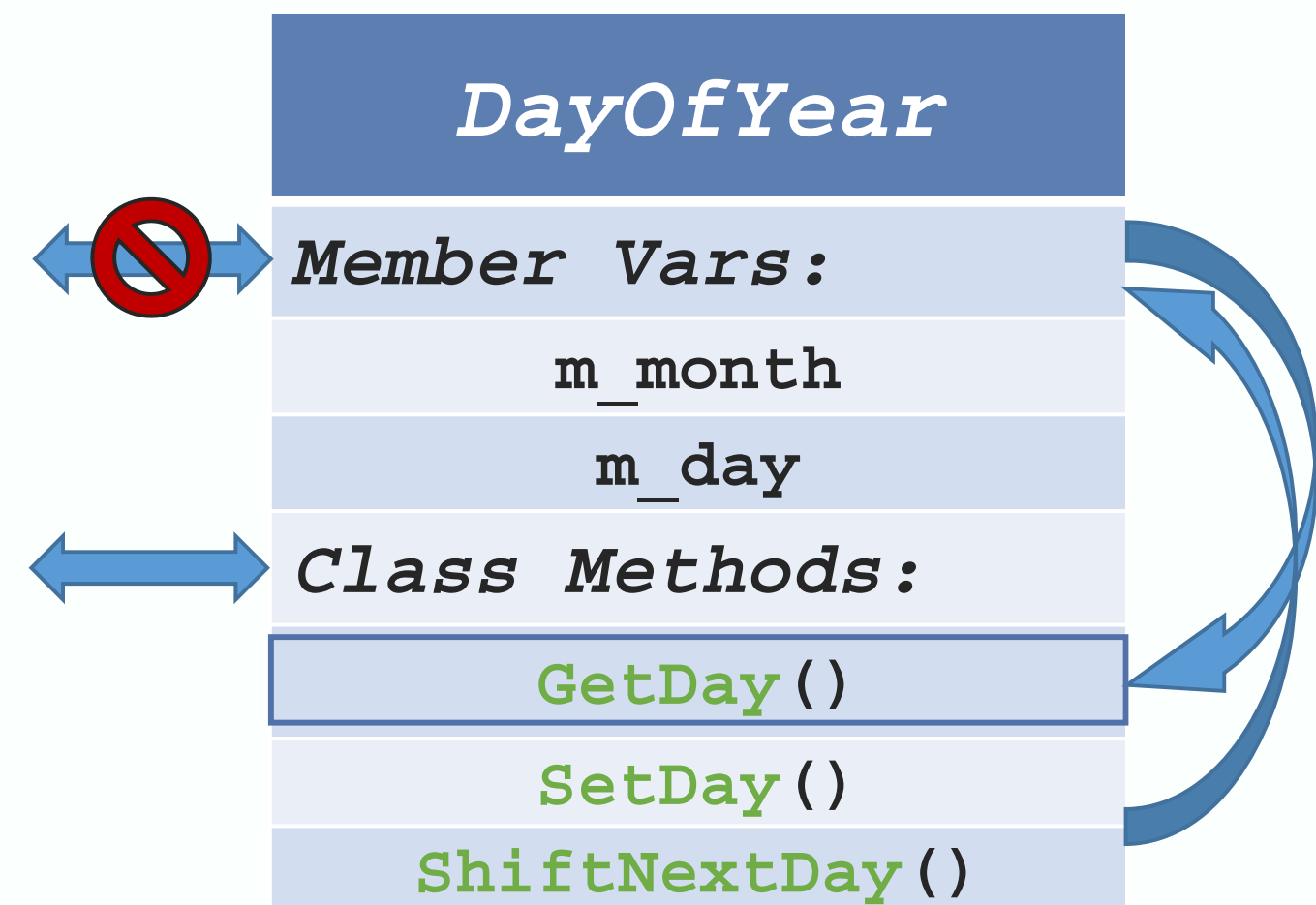
- *Accessor* Member Functions.

An Object-Interface to read Member Data.
Typically: “**GetMember**” Functions.

- *Mutator* Member Functions.

An Object-Interface to change Member Data.

Data manipulation, or “**SetMember**” Functions, based on application.



Protection Mechanisms

Accessor & Mutator Functions:

Object needs to “do something” with its data !

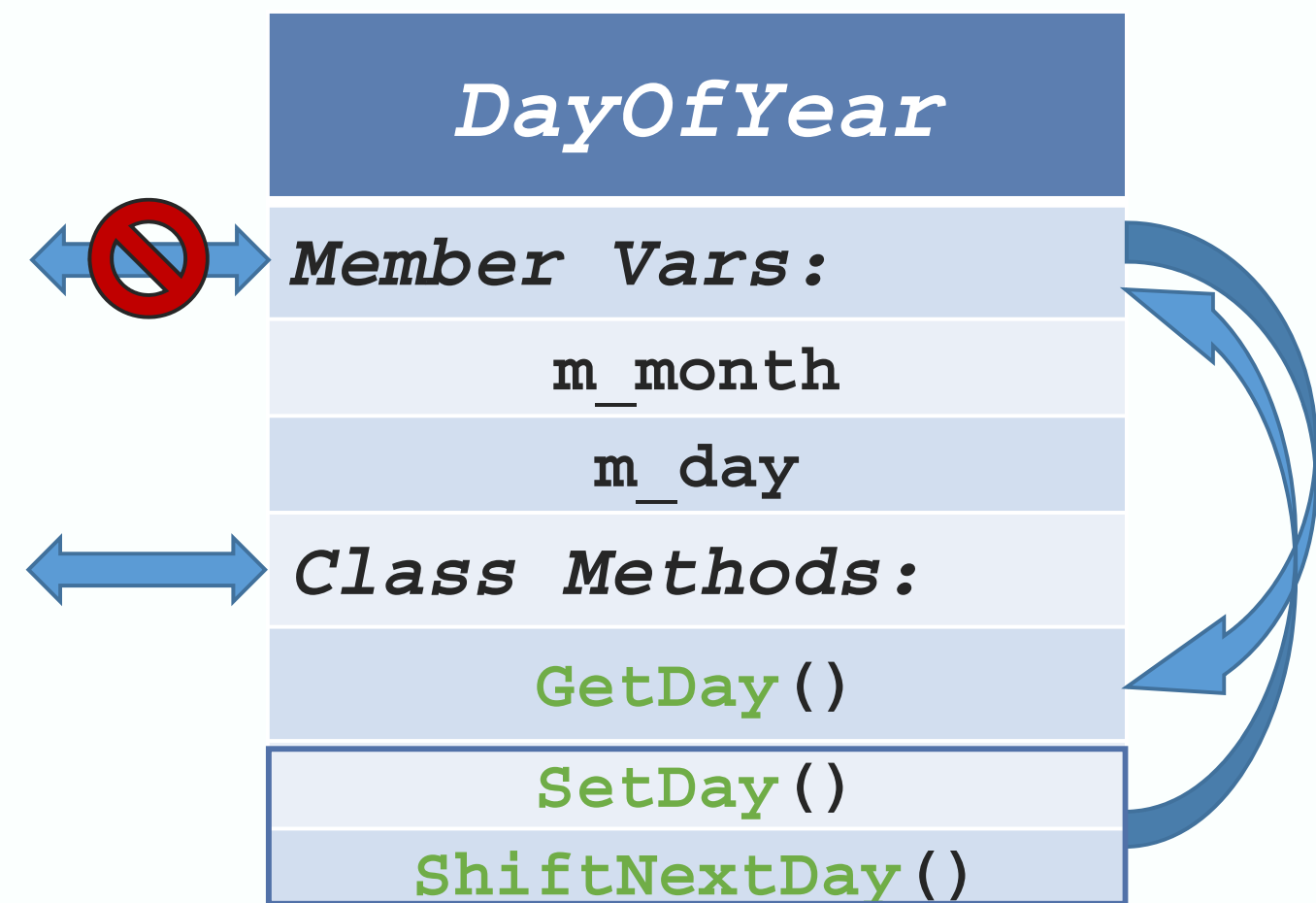
➤ *Accessor* Member Functions.

An Object-Interface to read Member Data.
Typically: “**GetMember**” Functions.

➤ *Mutator* Member Functions.

An Object-Interface to change Member Data.

Data manipulation, or “**SetMember**” Functions, based on application.



Protection Mechanisms

Accessor & Mutator Functions:

Object needs to “do something” with its data !

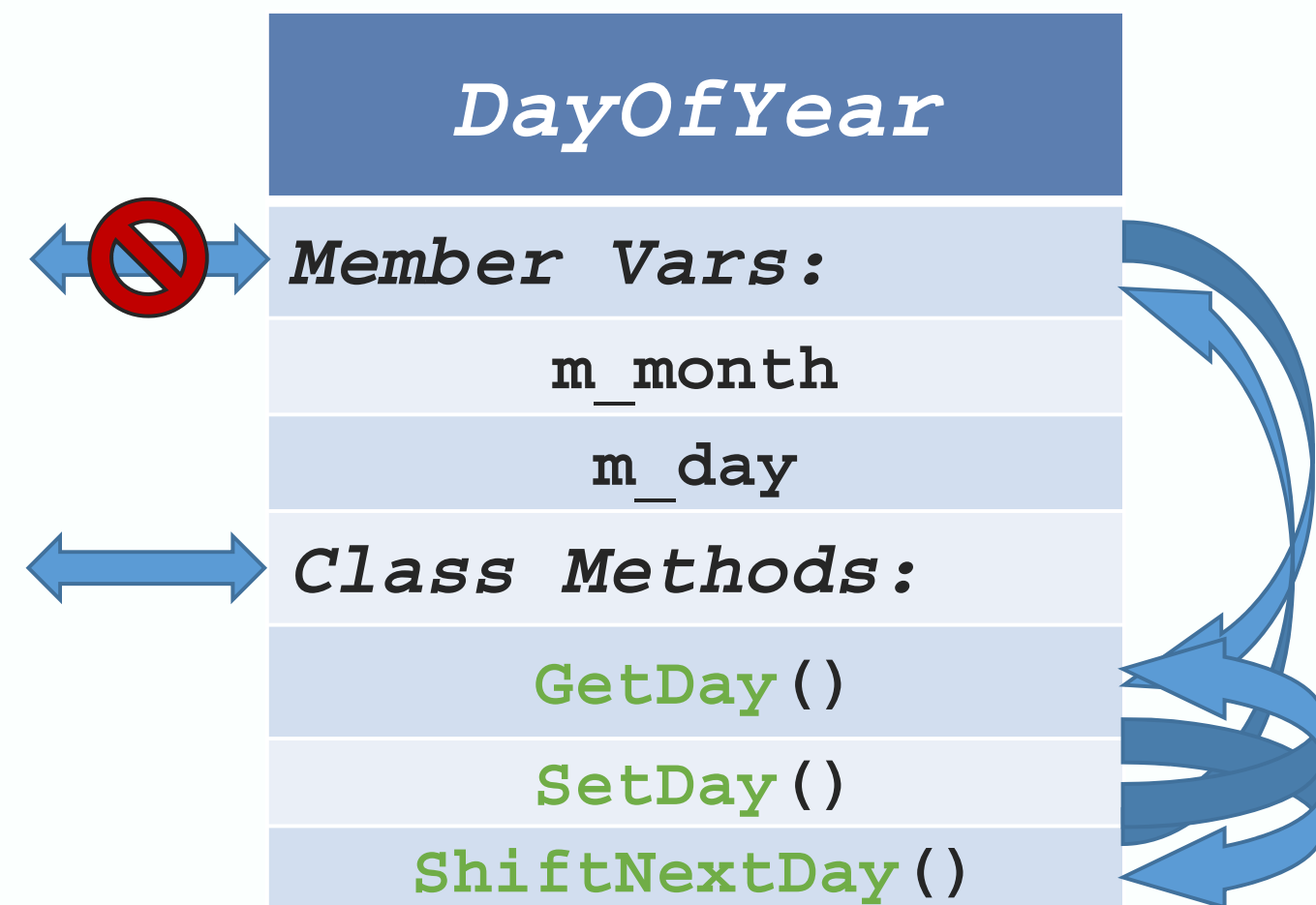
- **Accessor** Member Functions.

An Object-Interface to read Member Data.
Typically: “**GetMember**” Functions.

- **Mutator** Member Functions.

An Object-Interface to change Member Data.

Data manipulation, or “**SetMember**” Functions, based on application.



Object-Oriented Design (a correlation to Classes)

Thinking Objects / thinking with Objects:

- Focus on programming-style changes.
Before ➡ Algorithms at center stage.
OOP ➡ Data is the focal point.

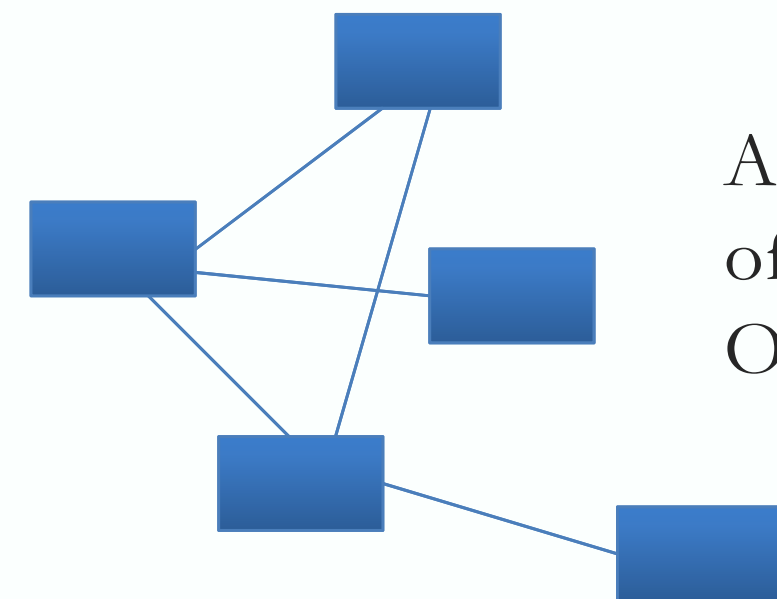
Algorithms still exist (of course):

- They focus on their data.
- Are “made” to “fit” the data.

Designing software solutions:

- Define variety of objects and how they interact.

A new type of Program Structure



A collection
of interacting
Objects

Object-Oriented Design (a correlation to Classes)

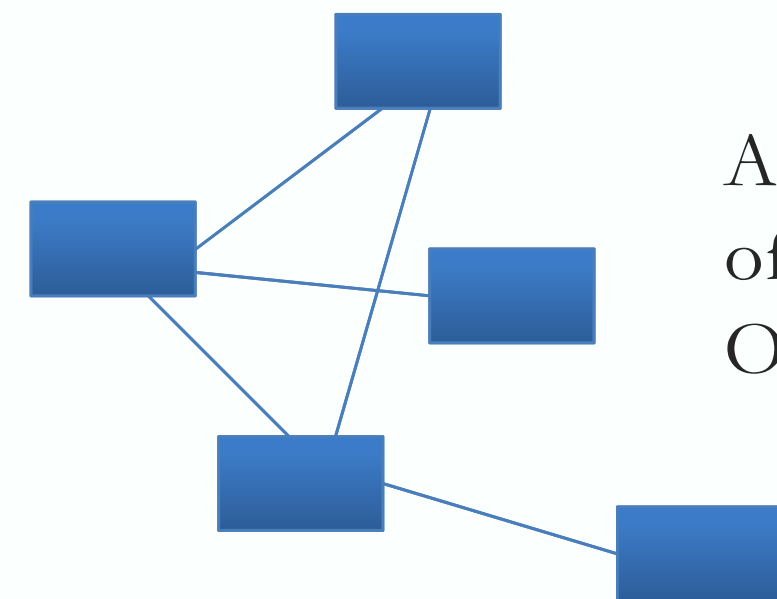
Thinking Objects / thinking with Objects:

- Focus on programming-style changes.
Before ➡ Algorithms at center stage.
OOP ➡ Data is the focal point.

Create large and powerful software systems from tiny components.

- Split things up into manageable pieces.
- Somewhat of a bottom up approach
(define little pieces that can be used to compose larger pieces).

A new type of Program Structure



A collection
of interacting
Objects

CS-202

Time for Questions !