

7

CSE 1325

Week of 10/05/2020

Instructor : Donna French

Object Oriented Programming

Intro to OOP

Going forward...

We are still learning how to program in C++

We will now add studying OO concepts and how to apply them in C++

Other OO languages will have ways of doing almost everything we will do – you are here to learn how to do them in C++

Intro to OOP

- CSE 1310
 - Teaching you how to program
 - Get you in the mindset of a programmer
- CSE 1320
 - Learning more advanced programming
 - What goes on behind the scenes (memory, pointers, debugging) using C
- CSE 1325
 - Learning a different type of programming
 - Objects, inheritance, encapsulation using C++

CSE 1310 and 1320 focused on procedural programming.

CSE 1325 will focus on OO programming

OOP

In traditional programming, programs are basically lists of instructions to the computer that define data and then work with that data.

Data and the functions that work on that data are separate entities that are combined together to produce the desired result.

Because of this separation, traditional programming often does not provide a very intuitive representation of reality.

OOP

It's up to the programmer to manage and connect the properties (variables) to the behaviors (functions) in an appropriate manner. This leads to code that looks like this:

```
driveTo(you, work);
```

A function called `driveTo` that takes `you` and `work` as parameters.

OOP

Object-oriented programming (OOP) provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

This leads to code that looks more like this:

```
you.driveTo(work) ;
```

You have the ability to drive to work...

OOP

Rather than being focused on writing functions, we focus on defining objects that have a well-defined set of behaviors.

This is why the paradigm is called “object-oriented”.

OOP allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability.

Simple Analogy – The Automobile as an Object

Suppose you want to drive a car and make it go faster by pressing its accelerator pedal.

What must happen before you can do this?

Step 1 - *design* the car

Simple Analogy – The Automobile as an Object

A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. A **class** is like a blueprint in that it is the template for any object created from it.

These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car.

This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Simple Analogy – The Automobile as an Object

Before you can drive a car, it must be *built* from the engineering drawings that describe it.

A completed car has an *actual* gas pedal to make the car go faster.

But the car won't accelerate on its own, so the driver must *press* the gas pedal to accelerate the car.

Simple Analogy – The Automobile as an Object

Gas pedal in our automobile

hides the mechanisms of making the car go faster from the driver

Brake pedal

hides the mechanisms of making the car stop from the driver

Function (also called a Method)

- houses the program statements that actually perform a task
- hides these statements from its user

Member functions

`get()`, `put()`, `getline()`, `size()`, `str()`

Simple Analogy – The Automobile as an Object

A car must be built from its drawings/blueprints before you can drive it.

An *object* must be built from a class before it can be used.

This building process is called *instantiation*.

An object is an instance of its class.

A class can be used many times to build many objects (more than one car is built from a drawing)

Simple Analogy – The Automobile as an Object

Pressing on the gas pedal sends a *message* to the car.

Pressing on the brake sends a different *message* to the car resulting in a different action happening.

Calling a member function is sending a *message* to an object.

Simple Analogy – The Automobile as an Object

What color is your car? What make is your car? What year is your car?

How many miles are on your car? How much gas is in the tank?

These are all *attributes* of your car. Every car not only has functions (gas pedal, brake pedal) but each one also has a unique set of *attributes*. My car knows how many miles are on it, but it does not know how many miles are on the car sitting next to it.

An object's attributes are defined in its class's **data members**.

Car

Attributes?

Functions?

Pencil Machine

Attributes

inventoryLevel

changeLevel

pencilCost

Functions

buyPencil

displayMoney

showInventoryLevel

showChangeLevel

```
struct DateStruct
{
    int year;
    int month;
    int day;
};
```

```
Today is 2019/9/23
Halloween is 2019/10/31
```

```
14 void print(DateStruct date)
15 {
16     cout << date.year << "/"
17     << date.month << "/" << date.day;
18 }
19
20 int main()
21 {
22     DateStruct today{2019,9,23};
23
24     cout << "Today is ";
25     print(today);
26     cout << endl;
27
28     today.month = 10;
29     today.day = 31;
30
31     cout << "Halloween is ";
32     print(today);
33     cout << endl;
34
35     return 0;
36 }
```

Why was the choice made to put the endl on a line by itself rather than inside the print() function?

Class

In the world of object-oriented programming, we want our types to not only hold data, but provide functions that work with the data as well.

In C++, this is typically done via the **class** keyword.

Using the **class** keyword defines a new user-defined type called a class.

Struct vs Class

```
struct DateStruct  
{  
    int year;  
    int month;  
    int day;  
};
```

```
class DateStruct  
{  
    public :  
        int year;  
        int month;  
        int day;  
};
```

Class

```
class DateClass
{
    public :
        int year;
        int month;
        int day;

    void print()
    {
        cout << year << "/"
              << month << "/"
              << day;
    }
};
```

```
int main()
{
    DateClass today{2019,9,23};

    cout << "Today is ";
    today.print();
    cout << endl;

    today.month = 10;
    today.day = 31;

    cout << "Halloween is ";
    today.print();
    cout << endl;

    return 0;    Today is 2019/9/23
                Halloween is 2019/10/31
}
```

Class vs Struct

The struct version of the program is pretty similar to the class version.

However, there are a few SIGNIFICANT differences.

In the `DateStruct` version of `print()`, the struct itself was passed to the `print()` function as the first parameter. Otherwise, `print()` wouldn't know which `DateStruct` to use. The function then had to reference that parameter inside the function explicitly.

```
DateStruct today{2019,9,23};
```

```
print(today);
```

```
cout << date.year << "/" << date.month << "/" << date.day;
```

Class vs Struct

```
void print()  
{  
    cout << year << "/" << month << "/" << day;  
}
```

Member functions work slightly differently - all member function calls must be associated with an object of the class.

When “`today.print()`” is called, the compiler calls the `print()` member function that is associated with the `today` object.

So when we call “`today.print()`”, the compiler interprets `day` as `today.day`, `month` as `today.month`, and `year` as `today.year`.

If we called “`tomorrow.print()`”, `day` would refer to `tomorrow.day` instead.

```
int main()  
{  
    DateClass today{2019,9,23};  
    DateClass tomorrow{2019,9,24};  
  
    cout << "Today is ";  
    today.print();  
    cout << endl;  
  
    cout << "Tomorrow is ";  
    tomorrow.print();  
    cout << endl;  
  
    return 0;  
}
```

Today is 2019/9/23
Tomorrow is 2019/9/24

Class vs Struct

The associated object is essentially implicitly passed to the member function.

For this reason, it is often called the implicit object.

The key point is that with non-member functions, we have to pass data to the function to work with.

With member functions, we can assume we always have an implicit object of the class to work with.

C++ Structs vs C Structs

C++ Struct

- default access is public
- can have member functions
- can directly initialize structure members
- can use `static`
- can have a constructor
- `sizeof()` empty struct will be 1

C Struct

- default access is public
- no member functions
- cannot directly initialize structure members
- cannot use `static`
- cannot have a constructor
- `sizeof()` empty struct will be 0

C++ Class vs C++ Structs

C++ Class

- members of a class are private by default
- when deriving a class, default access specifier is private.

C++ Struct

- members of a struct are public by default
- default access-specifier for a struct is public.

C++ Classes vs C Structs

C++ Class

- default access is private
- constructor/destructor run automatically
- member functions
- can use operator overloading
- inheritance

C Struct

- default access is public
- must be called explicitly
- no member functions

Intro to OOP

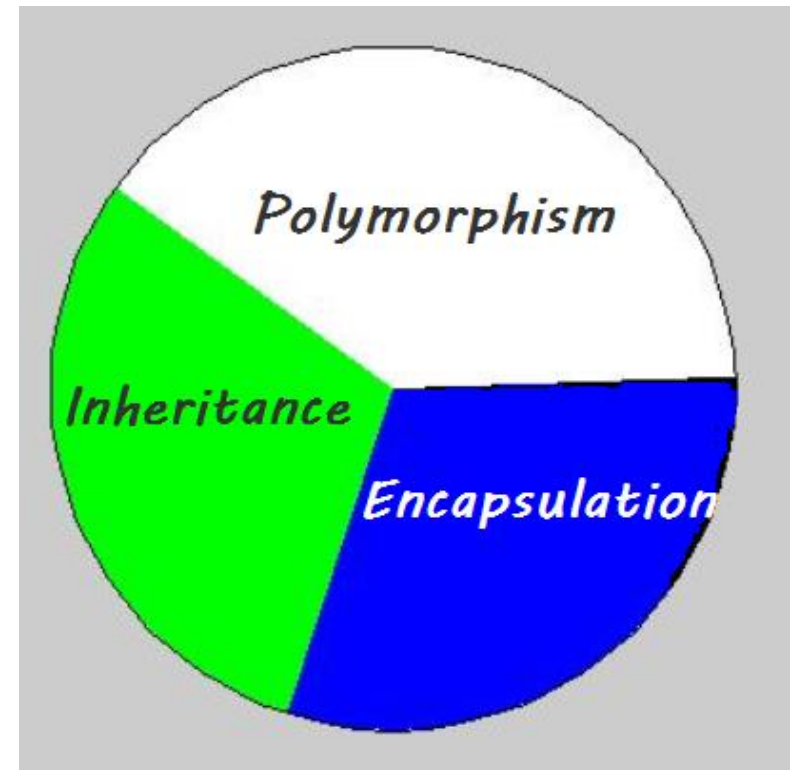
Three main concepts of Object Oriented Programming

OO PIE

Polymorphism

Inheritance

Encapsulation



OOP Vocabulary

Polymorphism

describes the ability of different objects to be accessed by a common interface

Inheritance

describes the ability of objects to derive behavior patterns from other objects

These objects can then customize and add new unique characteristics

Encapsulation

describes the ability of objects to maintain their internal workings independently from the program they are used in

Classes **encapsulate** attributes and member functions into objects created from those classes

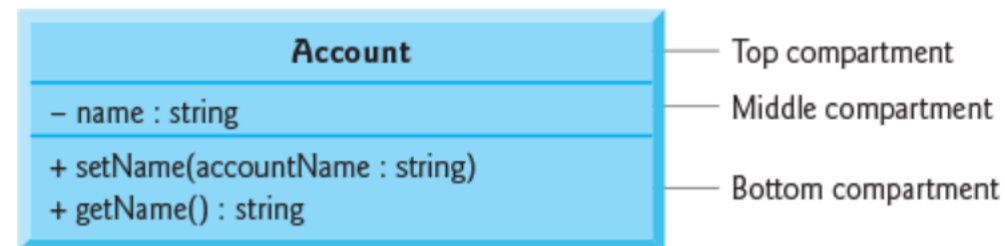
OOP Vocabulary

UML

Unified Modeling Language

widely used graphical scheme for modeling object-oriented systems

We will focus only on class diagrams



Creating a Bank Account Class

Class Account

Data Members

name

balance

Member functions

getBalance

depositFunds

withdrawFunds

Each class you create becomes a new type you can use to create objects.


```
#include <iostream>
#include <string>
#include "Account.h"
```

Include file for file
containing class
definition

```
using namespace std;
```

```
int main()
```

```
    string NewAccountName;
```

```
    Account MyBankAccount;
```

```
    cout << "My bank account's name is " << MyBankAccount.getName();
```

```
    cout << "\nEnter a new name for the bank account ";
```

```
    getline(cin, NewAccountName);
```

```
    MyBankAccount.setName(NewAccountName);
```

```
    cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;
```

```
    return 0;
```

```
}
```

TestBankAccount.cpp

Instantiate the object
MyBankAccount from class
Account

Member
function
getName()

Member function setName()

Member
function
getName()

```
class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};
```

Account.h

```
using namespace std;
```

is not used inside classes – using it in a class would force its usage in any program using the class – not the job of a class

Class definition

```
class ClassName
{

};
```

Each class gets its own header file.

```

class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};

```

setName

- member function
- return type of void
- one parameter of type string
- set data member name to the passed in accountName

getName

- member function
- return type of std::string
- no parameters
- returns data member name

Member function `getName()` is declared as `const` because, in the process of returning `name`, the function does not, and should not, modify its object `Account`.

With the `const` in place, the compiler would issue a warning if any code that modified `name` in that function was accidentally added.

```
class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};
```

name

- data member
- type `std::string`
- default value is empty string ""

Data members are declared inside a class definition but outside the bodies of the class's member functions.

Programming style – list data members at the end of the class to make them easier to find and to group together.

```
class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};
```

```
public
private
```

- keywords
- access specifier
 - always followed by a :

```
private
```

Since `name` appears after the access specifier `private`, `name` is only accessible to class `Account`'s member functions.

This is known as **data hiding** —the data member name is *encapsulated* (hidden) and can be used *only* in class `Account`'s `setName()` and `getName()` member functions.

Most data member declarations appear after the `private:` access specifier.

```
class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};
```

```
public
private
```

- keywords
- access specifier
 - always followed by a :

```
public
```

Data members or member functions listed after access specifier `public` (and before the next access specifier if there is one) are “available to the public.” They can be used by other functions in the program (such as `main`), and by member functions of other classes (if there are any).

By default, everything in a class is private, unless you specify otherwise.

Once you list an access specifier, everything from that point has that access until you list another access specifier.

Programming Style

List public only once, grouping everything that's public.

List private only once, grouping everything that's private.

The access specifiers public and private may be repeated, but this is unnecessary and can be confusing.

Making a class's data members private and member functions public makes debugging more productive because problems with data manipulations are localized to the member functions.

An attempt by a function that's not a member of a particular class to access a private member of that class is a compilation error.

```
student@cse1325:/media/sf_VM$ more makefile
#Donna French
#makefile for C++ program
SRC = TestBankAccount.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11
```

```
all : $(EXE)
```

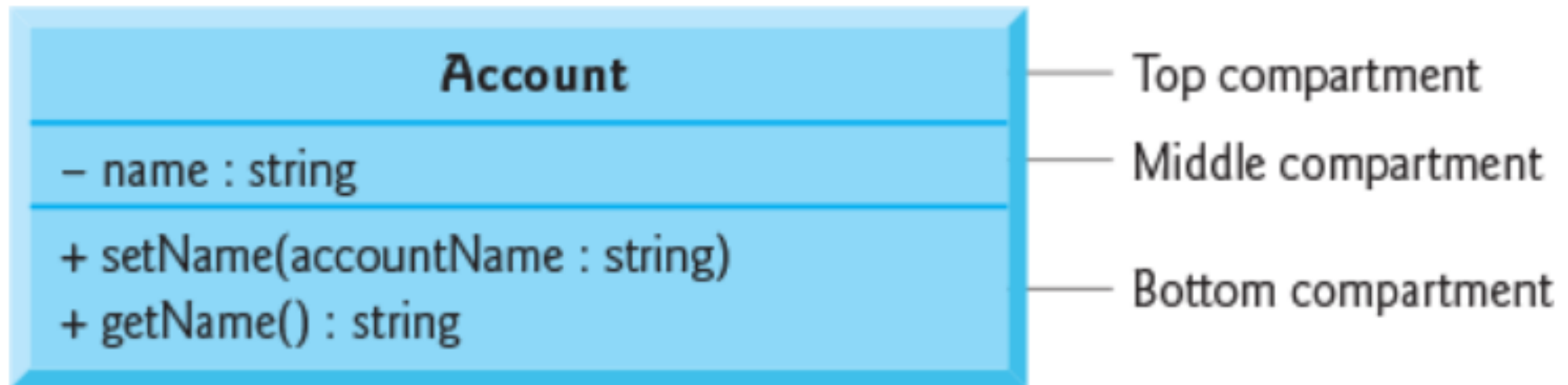
```
$(EXE): $(OBJ)
        g++ $(CFLAGS) $(OBJ) -o $(EXE)
```

```
$(OBJ) : $(SRC)
        g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
g++ -g -std=c++11 TestBankAccount.o -o TestBankAccount.e
student@cse1325:/media/sf_VM$ ./TestBankAccount.e
My bank account's name is
Enter a new name for the bank account My Bank Account
My bank account has been renamed My Bank Account
student@cse1325:/media/sf_VM$ █
```


Class Diagram

widely used graphical scheme for modeling objects in OOP

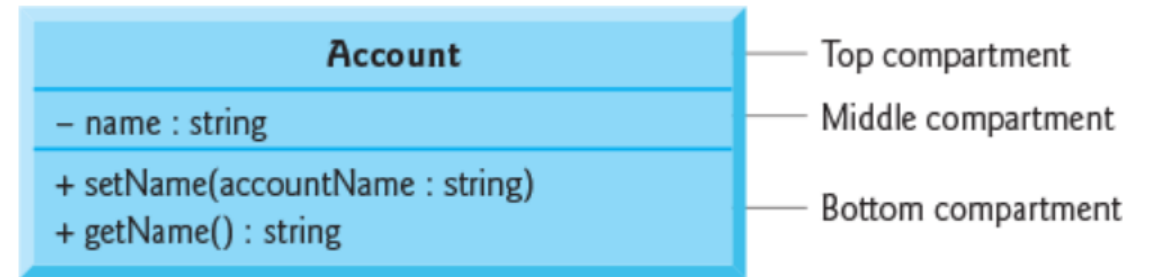


```

class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};

```



Top compartment

Account

Class name

Centered in the compartment

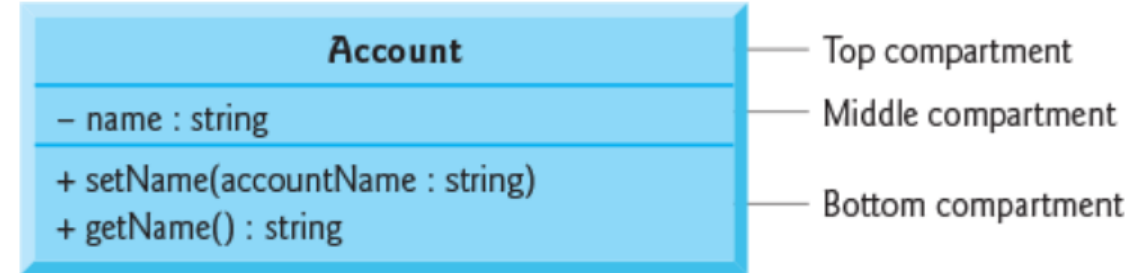
Account.h

```

class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};

```



Middle compartment

- name : string

- Class's attribute
name
- Minus sign indicates that the attribute is private
- Following the attribute name are a *colon* and the *attribute type*

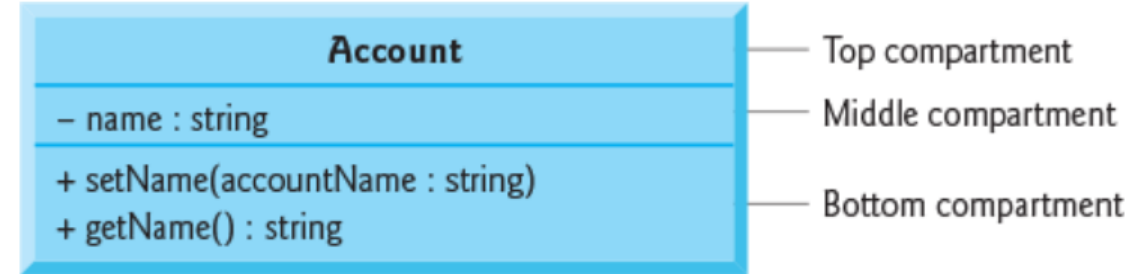
: string

```

class Account
{
    public :
        void setName(std::string accountName)
        {
            name = accountName;
        }

        std::string getName() const
        {
            return name;
        }
    private :
        std::string name;
};

```



Bottom compartment

Class's operations

Plus sign indicates that the operation is public

The () after the operation name contain

MFN (ParameterName : ParameterType) ReturnType

```

setName(accountName : string)
getName() : string

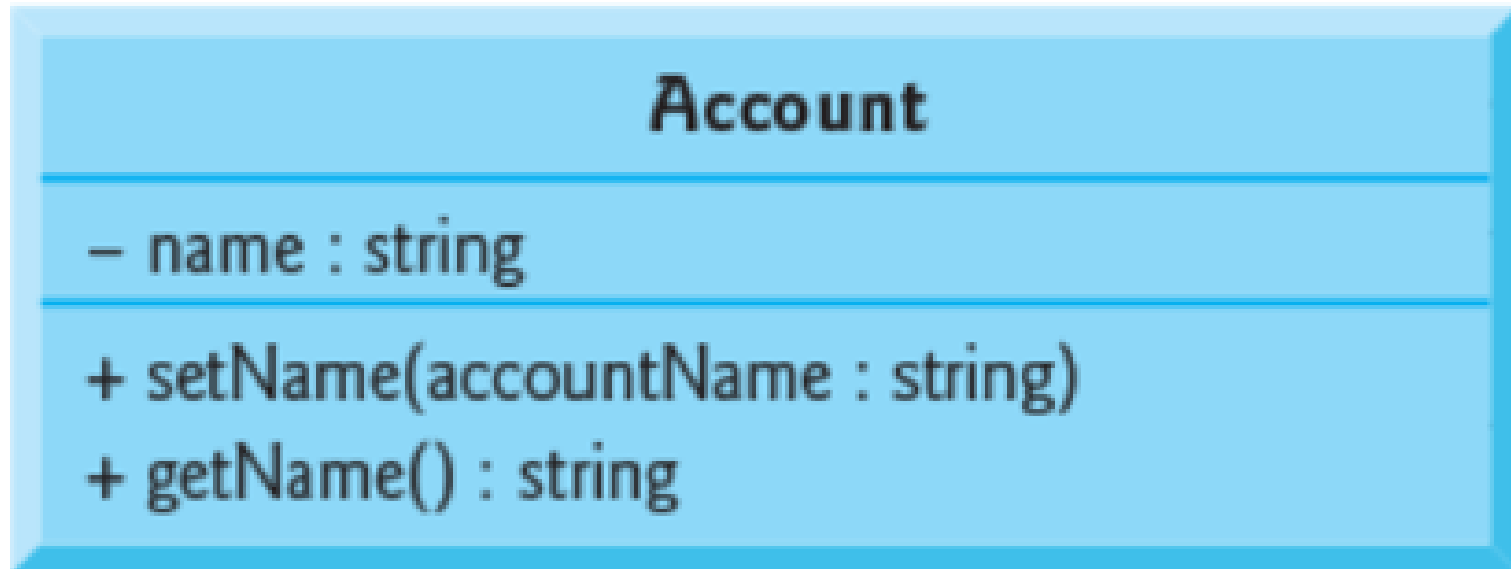
```

No return type - void

No parameters

```
class Account
{
    public :
        void setName(std::string accountName)
        {
        }

        std::string getName() const
        {
        }
    private :
        std::string name;
};
```



Circle
- radius : double = 0
+ getRadius() : double
+ setRadius(new_radius : double = 0)
+ calculateCircumference() : double

```
class Circle
{
    public :
        double getRadius()
        {
        }
        void setRadius(double new_radius=0)
        {
        }
        double calculateCircumference()
        {
        }
    private :
        double radius = 0;
};
```

```
class Student
```

```
{
```

```
    public :
```

```
        std::string getStudentID()
```

```
        {
```

```
        }
```

```
        void setStudentID(std::string newStudentID)
```

```
        {
```

```
        }
```

```
        std::string getNetID()
```

```
        {
```

```
        }
```

```
    private :
```

```
        std::string studentID;
```

```
        std::string netID;
```

```
        std::string emailAddress;
```

```
};
```

Student

- studentID : string

- netID : string

- emailAddress : string

+ getStudentID() : string

+ setStudentID(newStudentID : string)

+ getNetID() : string

+ setNetID(newNetID : string)

+ getEmailAddress() : string

+ setEmailAddress(newEmailAddress : string)

```
class DateClass
{
```

```
    public :
```

```
        int year;
```

```
        int month;
```

```
        int day;
```

```
    void print()
```

```
    {
```

```
        cout << year << "/"
```

```
        << month << "/" << day;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    DateClass today{2019,9,25};
```

```
    cout << "Today is " << today.month << today.day
```

```
    << today.year << endl;
```

```
    today.print();
```

```
    return 0;
```

```
}
```

```
student@cse1325:/media/sf_VM$ ./pvspDemo.e
```

```
Today is 9252019
```

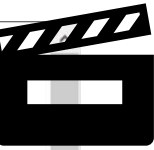
```
2019/9/25student@cse1325:/media/sf_VM$
```

Data members are public; therefore,
can be accessed directly.

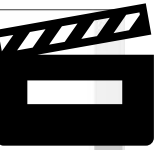

```
class Date {
public:
    // ...
private:
    // ...
};

int main() {
    Date d;
    cout << d.getDate();
    today = d.getDate();
    return 0;
}
```





```
1 // public vs private Demo
2
3 #include <iostream>
4
5 using namespace std;
6
7 class DateClass
8 {
9     public :
10         void print()
11         {
12             cout << year << "/"
13                 << month << "/" << day;
14         }
15         void setDate(int mnth, int dy, int yr)
16         {
17             year = yr;
18             month = mnth;
19             day = dy;
20         }
21     private :
22         int year;
23         int month;
24         int day;
25 };
26
27 int main()
28 {
```



```
4
5 using namespace std;
6
7 class DateClass
8 {
9     public :
10     void print()
11     {
12         cout << year << "/"
13             << month << "/" << day;
14     }
15     void setDate(int mnth, int dy, int yr)
16     {
17         year = yr;
18         month = mnth;
19         day = dy;
20     }
21     private :
22     int year;
23     int month;
24     int day;
25 };
26
27 int main()
28 {
29     DateClass today;
30
31     today.setDate(9,25,2019);
```

```
class PVSP
{
    public :
        int x;
        char y;
        double z;
    private :
        string a;
        float b;
        long c;
};
```

```
int main()
{
    PVSP QuizMe;

    QuizMe.x = 1;
    QuizMe.a = "Quiz";
    QuizMe.y = 'A';
    QuizMe.b = 1.23;
    QuizMe.z = 1.23;
    QuizMe.c = 123;

    return 0;
}
```

Which lines will
compile?

Encapsulation

You will begin to notice that objects tend to have mostly private data members and public member functions.

Why?

Take the example of the electronic devices that surround us every day.

They have simple interfaces that allows you to perform actions without know the details behind those actions.

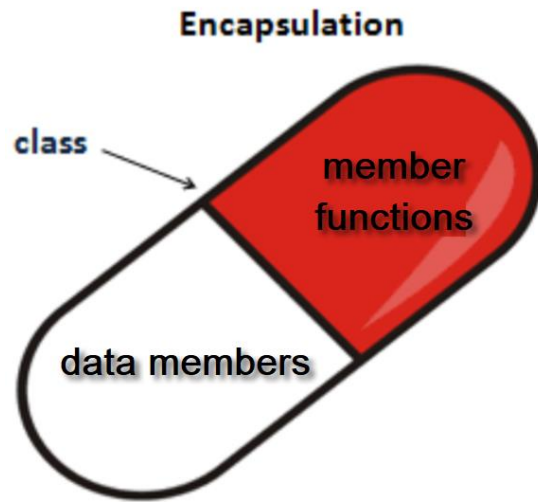
Encapsulation

The separation of interface and implementation is extremely useful because it allows us to use objects without understanding how they work.

This vastly reduces the complexity of using these objects and increases the number of objects we are capable of interacting with.

This same principle is applied to programming – separating implementation from interface.

Generally, data members are made private (hiding implementation details) and member functions are made public (giving the user an interface).



Encapsulation

Classes wrap attributes and member functions into objects created from those classes – an object's attributes and member functions are intimately related.

Objects may communicate with one another, but they are not normally allowed to know how other objects are implemented.

Encapsulation is the technique of information hiding - implementation details are hidden within the objects themselves.

Encapsulation

Benefits of Encapsulation

Encapsulated classes are easier to use and reduce the complexity of programs.

Encapsulated classes help protect your data and prevent misuse

Encapsulated classes are easier to debug

Encapsulation

Access Functions

Getter

A function that returns the value of a private variable

Setter

A function that changes the value of a private variable

```
class DateClass
```

```
{
```

```
    public :
```

```
        void print()
```

```
{
```

```
    cout << year << "/"
```

```
    << month << "/" << day;
```

```
}
```

```
    private :
```

```
        int year;
```

```
        int month;
```

```
        int day;
```

```
};
```

```
int main()
```

```
{
```

```
    DateClass today{2019,9,25};
```

```
    cout << "Today is " << today.month << today.day
```

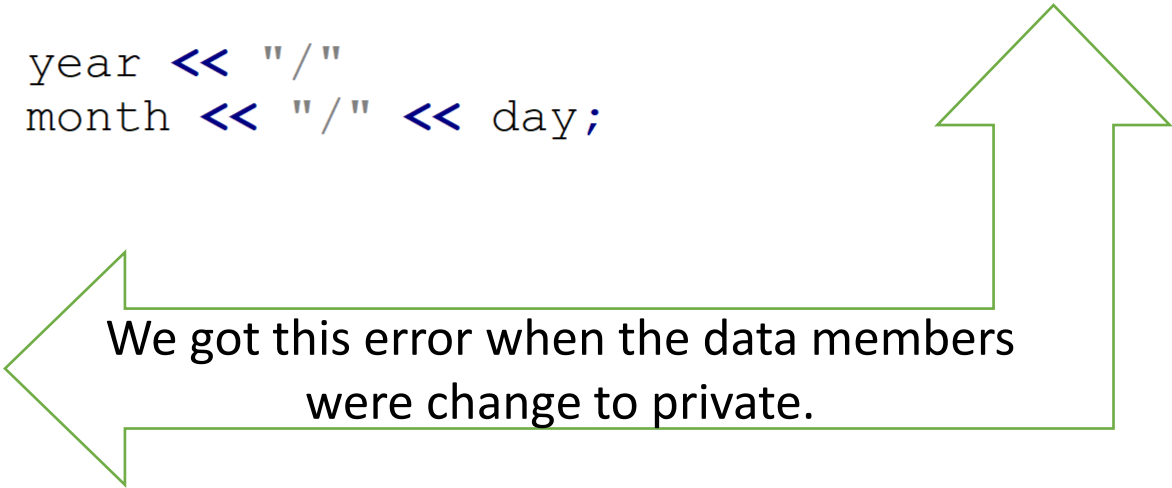
```
    << today.year << endl;
```

```
    today.print();
```

```
    return 0;
```

```
}
```

```
classDemo.cpp: In function 'int main()':  
classDemo.cpp:24:30: error: no matching function  
for call to 'DateClass::DateClass(<brace-  
enclosed initializer list>)'  
    DateClass today{2019,9,23};
```



We got this error when the data members
were change to private.

If you can't directly
access a variable
(because it's private),
you shouldn't be able to
directly initialize it.

Constructors

A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.

Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used.

Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class (with the same capitalization)

Initializing Objects with Constructors

```
My bank account's name is  
Enter a new name for the bank account CSE 1325's Bank Account  
My bank account has been renamed CSE 1325's Bank Account
```

Our initial bank account name prints out as blank because, when our object `MyBankAccount` was created, `name` was initialized to the empty string.

What if you want the freshly created object to have a name already?

A class can define a **constructor** that specifies *custom initialization* for objects of that class.

Initializing Objects with Constructors

Constructor

- special member function

- must have the same name as the class

- have no return type (not even void)

- C++ requires a constructor call when *each* object is created

- ideal point in program to initialize an object's data members

- can have parameters

- A constructor's *parameter list* specifies pieces of data required to initialize an object
- usually public