Classes and Structures

Computer Programming for Engineers (DASF003-41)

Instructor:

Sungjae Hwang (jason.sungjae.hwang@gmail.com)

This Week

- **Structures**
  - Structure types
  - Structures as function arguments
  - Initializing structures
- **Classes**
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes
  - Object oriented programming paradigm

STRUCTURES

Structures

- **2nd aggregate data type: struct**
  - aggregate meaning "grouping"
  - Structure: heterogeneous collection of values of different types
  - Recall array: homogeneous collection of values of same type

- **Treated as a new data type**
  - Major difference: Must first "define" `struct` prior to declaring any variables

Structure Types

- Define `struct` globally (typically)
  - No memory is allocated
  - Just a "placeholder" for what our `struct` will "look like"

- Definition:

```
struct CDAccountV1 //name of new struct “type”, structure tag
{
   double      balance; //member names
   double      interestRate;
   int         term;
};
```

Declaring/Instancing Structure Variable

- Now we can declare variables of this new type:

```
CDAccountV1 account;
```

  - This is creating an instance of the structure.
  - Just like declaring simple types
  - Variable account now of type CDAccountV1

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

```
Struct CDAccountV1 account; //C version
```

Accessing Structure Members

Dot(.) Operator to access members

```
CDAccountV1 account;
account.balance; // member variable
account.interestRate;
account.term;
```

Called "member variables"

- The "parts" of the structure variable
- Different `structs` can have same name member variables

```
struct CDAccountV1
   double balance; //member names
};
struct CDAccountV2
   double balance; //member names
};
```

```
CDAccountV1 account1;
CDAccountV2 account2;

account1.balance;
account2.balance;
```

Structure Pitfalls

Semicolon after structure definition

- Semicolon (;) MUST exist in the end of the declaration:

```
struct WeatherData
{
   double temperature;
   double windVelocity;
}; //REQUIRED semicolon!
```

- Required since you "can" declare structure variables in this location

structure member names 1


```
#include <iostream>
#include <cmath>
using namespace std;

struct CDAccountV1 { // name of new struct "type"
  double  balance; // member names
  double  interestRate;
  int     term;
};

struct CDAccountV2 { // name of new struct "type"
  int   balance; // member names
  int   interestRate;
  int   term;
} account2;

int main()
{
  // C++
  CDAccountV1 account1;
  // C
  //struct CDAccountV1 account1;
```

structure member names 2


```
  account1.balance = 1000;
  account1.interestRate = 0.02;
  account1.term = 2;

  cout << "I have $" << account1.balance << " in my account." << endl;
  double rate1 = pow(1+account1.interestRate, account1.term);
  cout << "After " << account1.term << " years it will become $" << account1.balance * rate1 << "." <<
endl;

  // We can use the same names for member vars of different structs
  account2.balance = 2000;
  account2.interestRate = 0.02; // CHECK TYPE!
  account2.term = 5;

  cout << "I have $" << account2.balance << " in my account." << endl;
  double rate2 = pow(1+account2.interestRate, account2.term);
  cout << "After " << account2.term << " years it will become $" << account2.balance * rate2 << "." <<
endl;

  return 0;
}
```

Structures as Function Arguments

Passed like any simple data types

- Pass-by-value
- Pass-by-reference
  - Recommended, when the size of a structure is large
  - Avoids the redundant copy of the data
- Or combination

Can also be returned by a function

- The return-type is a structure type.
- The return statement in the function definition sends a structure variable back to the caller.

```
CDAccountV1 doubleInterest(CDAccountV1 acc);
```

Passing struct as argument (02_)


```
#include <iostream>
#include <cmath>
using namespace std;

struct CDAccountV1 { // name of new struct "type"
  double  balance; // member names
  double  interestRate;
  int     term;
};

void printAccountInfo(CDAccountV1 myAccount) {
  cout << "I have $" << myAccount.balance << " in my account." << endl;
  double rate = pow(1+myAccount.interestRate, myAccount.term);
  cout << "After " << myAccount.term << " years it will become $" << myAccount.balance * rate << "." <<
endl;
  // What happens when we modify the value of myAccount's member variables?
}

int main() {
  CDAccountV1 acc;
  acc.balance = 2000;
  acc.interestRate = 0.02;
  acc.term = 3;
  printAccountInfo(acc);
  return 0;
}
```

Initializing Structures

Aggregate initialization

- Declaration provides initial data to all three member variables

```
struct Date
{
   int month;
   int day;
   int year;
};
Date dueDate = {12, 31, 2003};
```

Non-static data member with initializer (C++11)

- This can be even simpler by providing default values in declaration.

```
struct Date
{
   int month = 12;
   int day = 31;
   int year = 2003;
};
```

Initializing Structures

Non-static data member with initializer (C++11)

- Not with c++98

```
struct Date
{
    int month = 12;
    int day = 31;
    int year = 2003;
};
```

```
yhoon@yhoonLabServer:~/Lecture/2020Fall/2020Fall_CPE/demo/lecture06/demo2020F$ g++ -std=c++98 3_struct_initialization.cc
3_struct_initialization.cc:8:15: warning: non-static data member initializers only available with '-std=c++11' or '-std=gnu++11'
    8 |    int month = 12;
      |              ^~
3_struct_initialization.cc:9:13: warning: non-static data member initializers only available with '-std=c++11' or '-std=gnu++11'
    9 |    int day = 31;
      |            ^~
3_struct_initialization.cc:10:14: warning: non-static data member initializers only available with '-std=c++11' or '-std=gnu++11'
   10 |    int year = 2003;
      |             ^~~~
yhoon@yhoonLabServer:~/Lecture/2020Fall/2020Fall_CPE/demo/lecture06/demo2020F$
```

struct initialization


```
#include <iostream>

using namespace std;

// Only from C++11
// g++ -std=c++98 ... shows an error or a warning
struct Date {
  int month = 12;
  int day = 31;
  int year = 2003;
};

int main()
{
  Date dueDate;
  cout << dueDate.month << endl;
  return 0;
}
```

CLASSES

Classes

Similar to structures

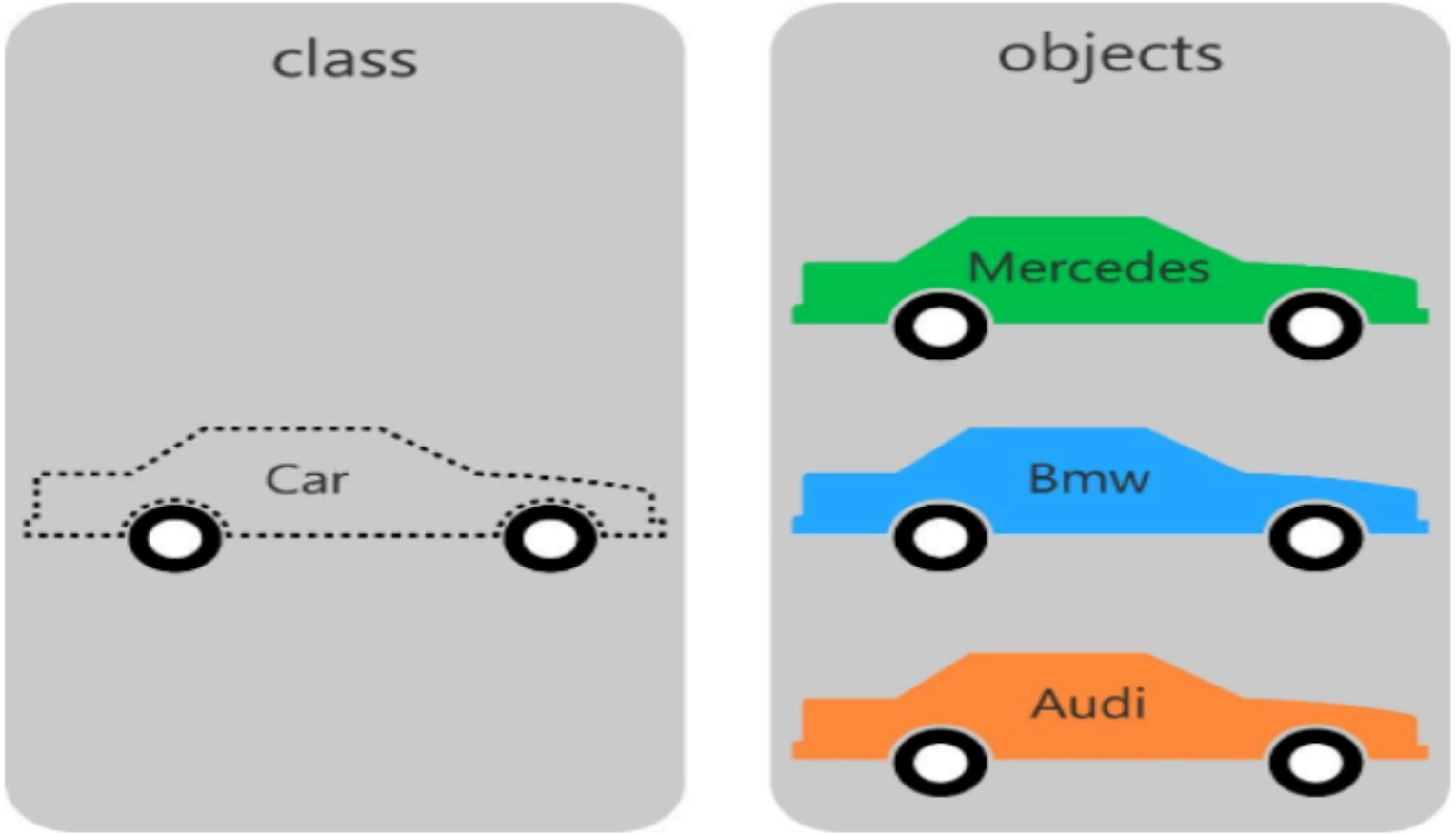
- Simply Adds member FUNCTIONS as well as member variables.

```
class DayOfYear  // name of new class type
{
public:
  int month; // member variable / attribute!
  int day;
  void output(); // member function! implementation elsewhere
};
```

Integral to object-oriented programming

- Focus on objects containing both data and operations.
- We can define object's behavior using the member functions.

Classes vs Objects



http://www.expertphp.in/article/php-classes-and-object

Declaring Objects

- **Declared same as all variables**
  - Predefined types, structure types
- **Example:**

```
DayOfYear today, birthday;
```

  - Declares two objects of class type DayOfYear
- **Objects include:**
  - Data members: month, day
  - Operations (member functions): output()

Class Member Access

Members accessed same as structures

```
today.month;
today.day;
today.output(); // invokes member function
```

Dot (.) and Scope Resolution (::) Operator

- Used to specify "of what thing" they are members
- Dot operator: specifies member of particular object
- Scope resolution operator: specifies what class the function definition comes from

Class Member Functions Definition

- Must define or "implement" class member functions
- Like other function definitions
  - Must specify class:

```
class DayOfYear  // name of new class type
{
   …
   int  day;
   int  month;
   void output(); // member function! implementation elsewhere
};

void DayOfYear::output(){
   …
}
```

  - :: is scope resolution operator
  - It instructs the compiler "what class" member is from.

`this` Pointer in Member Function

`this` pointer:

- Predefined pointer to the calling object itself

```
void DayOfYear::output(){ // void output(DayOfYear* const this)
   cout << this->month << endl;
   cout << this->day << endl;
}
object1.output();  // object1.output(&object1);
```

- '`->`': member dereference operator (the same as in C)

Scope conflict

- When member function parameter has the same name, we can only access the data member using `this`.

```
void DayOfYear::assign ( int month, int day )
{
   this->month = month;
   this->day = day;
}
```

member functions and this 1


```
#include <iostream>
using namespace std;
class DayOfYear { // name of new class type
  public:
    void output(); // member function! implementation elsewhere
    void assign1(int m, int d);
    void assign2(int month, int day);
    void assign3(int, int);
    int month;
    int day;
};

void DayOfYear::assign1(int m, int d) {
  month = m;
  day = d;
}

void DayOfYear::assign2(int month, int day) {
  this->month = month;
  this->day = day;
}
```

member functions and this 2


```
void DayOfYear::assign3(int month, int day){
  this->month = month;
  this->day = day;
}
void DayOfYear::output() {
  cout << month << "/" << day << endl;
}

int main() {
  DayOfYear birthday;

  birthday.month = 5;
  birthday.day = 11;
  birthday.output(); // invokes member function

  birthday.assign1(9,6);
  birthday.output();

  birthday.assign2(1,22);
  birthday.output();

  birthday.assign3(12,23);
  birthday.output();
}
```

Complete Class Example

Display 6.3 Class With a Member Function (1 of 5)

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear
//will be given in Display 6.4.
#include <iostream>
using namespace std;
class DayOfYear
{
public:
    void output( ); //Member function declaration
    int month;
    int day;
    /*Normally, member variables are private
    and not public, as in this example. This is
    discussed a bit later in this chapter./*
};
int main( )
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
```

Complete Class Example

Display 6.3 Class With a Member Function (2 of 5)

```
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
    cout << "Today's date is ";
    today.output( );
    cout << endl;
    cout << "Your birthday is ";
    birthday.output( );
    cout << endl;
    if (today.month == birthday.month && today.day == birthday.day)
      cout << "Happy Birthday!\n"
    else
      cout << "Happy Unbirthday!\n";
    return 0;
}
```

Complete Class Example

Display 6.3 Class With a Member Function (3 of 5)

```
//Uses iostream:
void DayOfYear::output( ) //Member function definition
{
    switch (month)
    {
      case 1:
          cout << "January "; break;
      case 2:
          cout << "February "; break;
      case 3:
          cout << "March "; break;
      case 4:
          cout << "April "; break;
      case 5:
          cout << "May "; break;
      case 6:
          cout << "June "; break;
      case 7:
          cout << "July "; break;
```

Complete Class Example

Display 6.3 Class With a Member Function (4 of 5)

```
        case 8:
            cout << "August "; break;
        case 9:
            cout << "September "; break;
        case 10:
            cout << "October "; break;
        case 11:
            cout << "November "; break;
        case 12:
            cout << "December "; break;
        default:
            cout << "Error in DayOfYear::output.";
    }
    cout << day;
}
```

Complete Class Example

Display 6.3 Class With a Member Function (5 of 5)

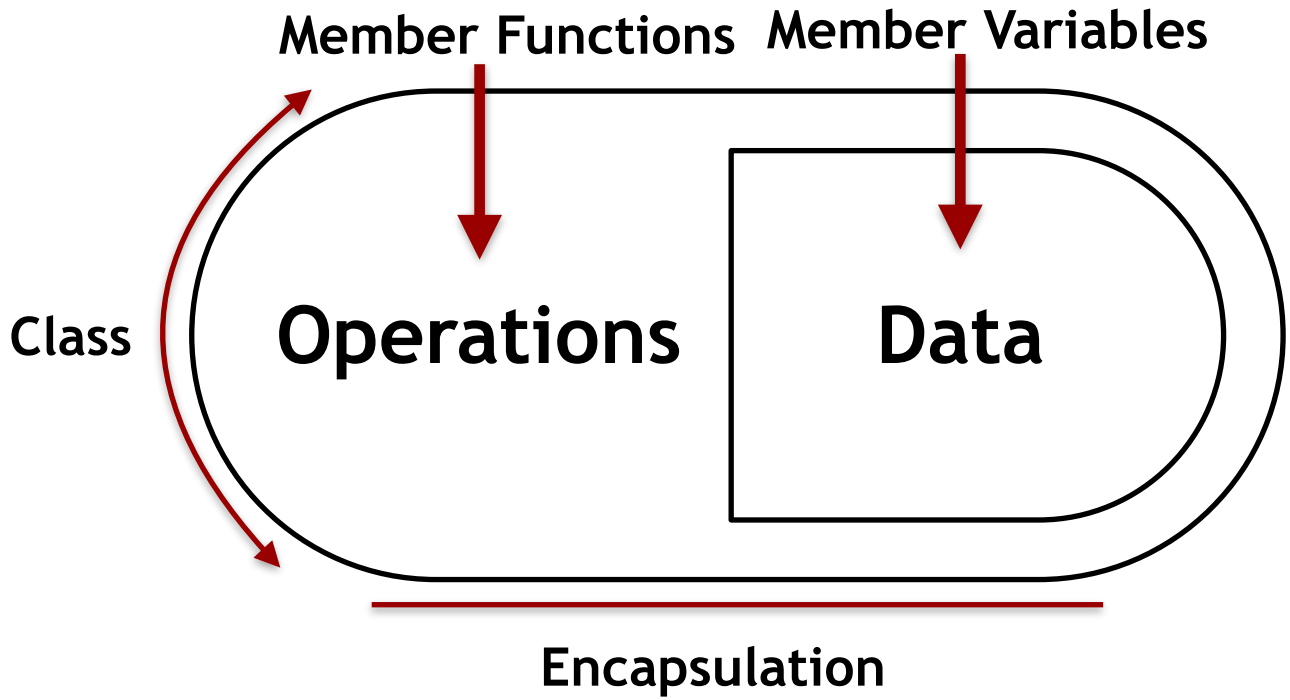
```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is October 15
Your birthday is February 21
Happy Unbirthday!
```

CLASS DETAILS

A Class's Place

- **Class is full-fledged type!**
  - Just like data types int, double, etc.
- **Can have variables of a class type**
  - We simply call them "objects"
- **Can have parameters of a class type**
  - Pass-by-value
  - Pass-by-reference
- **Can use class type like any other type!**

Encapsulation



- **Benefits of encapsulation**
  - Provide Information Hiding
  - Provide Abstraction (Hide detail implementation)
  - Prevent unnecessary access / invalid modification of data

Public and Private Members

- **Data in class almost always designated private in definition!**
  - Upholds principles of OOP
  - Hide data from user
  - Allow manipulation only via operations/member functions
- **Public items (usually member functions) are "user-accessible"**

Class Example

- Violate the principle of encapsulation

```
class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
  DayOfYear today;
  cout << "Enter month: ";
  cin >> today.month;
  cout <<"Enter day of Month: ";
  cin >> today.day;

  today.output();
```

```
//Member function definition
void DayOfYear::output( ){
    switch (month)
    {
      case 1:
          cout << "January “;
           break;
      case 2:
          cout << "February “;
           break;
      case 3:
          cout << "March “;
           break;
     …
     }

  cout << day;
```

Public and Private Qualifiers

- Modify previous example:

```
class DayOfYear
{
public: //can be directly accessed even from non-member functions
   void input();
   void output();
private:  //can be directly accessed only in member functions
   int month;
   int day;
};
```

- Data now private, while member functions are public

```
int main( )
{
  cin >> today.month;   // NOT ALLOWED!
  cout << today.day;    // NOT ALLOWED!
  today.output();      // ALLOWED!
}
```

Public and Private Style

- Can mix & match public & private
- More typically place public first
  - Allows easy viewing of portions that can be USED by programmers using the class
  - Private data is "hidden", so irrelevant to users

```
public:
  void output();
  void input();

private:
  int a;
  int b;
```

```
private:
  int a;
  int b;

public:
  void output();
  void input();
```

```
private:
  int a;
public:
  void output();
  void input();
private:
 int b;
```

public vs private (05_)


```
#include <iostream>
using namespace std;
class DayOfYear {
  public:
    void output(); // member function! implementation elsewhere
    void assign(int month, int day);
  private:
    int month;
    int day;
};
void DayOfYear::output() {
  cout << month << "/" << day << endl;
}
void DayOfYear::assign(int month, int day) {
  this->month = month;
  this->day = day;
}
int main() {
  DayOfYear birthday;
  // Illegal accesses to private member variables
  //birthday.month = 5;
  //birthday.day = 11;
  birthday.assign(5, 11);
  birthday.output(); // invokes member function
  return 0;
}
```

Accessor and Mutator Functions

- **Object needs to "do something" with its data**
- **Call accessor member functions**
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data
- **Mutator member functions**
  - Allow object to change data
  - Manipulated based on application

```
public:
  // Accessor Function!
  int getMonth();

  // Mutator Function!
  void setMonth();

private:
  int month;
```

`const`: Usages in Member Functions

- Accessors typically accompany const after:

```
class DayOfYear {
public:
   int get_day() const; // typical accessor definition
};
```

- Usages of `const`:

```
class DayOfYear{
   const int* const get_pointer_to_day() const;
};
```

  - first `const`: the value referenced by the pointer is constant (immutable)
  - second `const`: the pointer itself is constant
  - third `const`: modification to any class member variables are not allowed in the function.

const modifier 1


```
 1 #include<iostream>
 2 using namespace std;
 3
 4 class CPE {
 5    int studentNo;
 6
 7    public:
 8    CPE(int num = 0){
 9      studentNo = num;
10    }
11    int getNumOfStudent() const {
12       return studentNo;
13    }
14
15    int setNumOfStudent(int num) {
16       return studentNo = num;
17    }
18 };
19
20 int main() {
21    const CPE one;
22    CPE two;
23
24    //one.setNumOfStudent(52);
25    //two.setNumOfStudent(52);
26
27    cout << "The value using object d : " <<
one.getNumOfStudent();
28    cout << "\nThe value using object d1 : "
<< two.getNumOfStudent();
29    return 0;
30 }
```

const modifier 2


```
#include <iostream>
using namespace std;
int main() {
  int number7 = 7;
  int number8 = 8;
  int* const addr_const = &number7;
  const int* const_addr = &number7;
  cout << number7 << " has an address of " << addr_const << endl;
  cout << number7 << " has an address of " << const_addr << endl;

  // ERROR: To check the address of number8
  //addr_const = &number8;
  //cout << number8 << " has an address of " << addr_const << endl;
  const_addr = &number8;
  cout << number8 << " has an address of " << const_addr << endl;

  // Assigning a new value through pointers, addr_const points to number7
  *addr_const = 77;
  cout << "number7 is now " << *addr_const << endl;

  // ERROI: const_addr points to number8
  //*const_addr = 88;
  //cout << "number8 is now " << *const_addr << endl;
}
```

Limitation

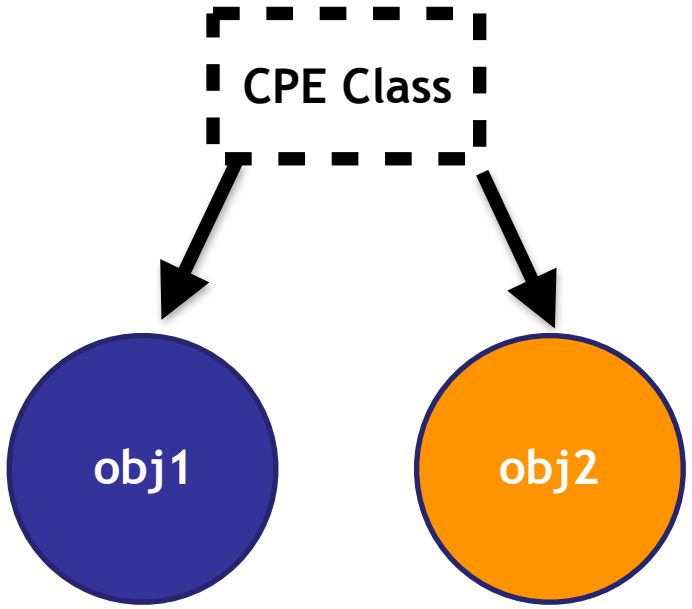
- Modify previous example:

```
class CPE
{
  public:
   int month;
};

int main(){
  CPE obj1;
  CPE obj2;

  obj1.month = 1;
  obj2.month = 2;

  Return 0;
}
```



obj1 != obj2

obj1.month != obj2.month

Limitation

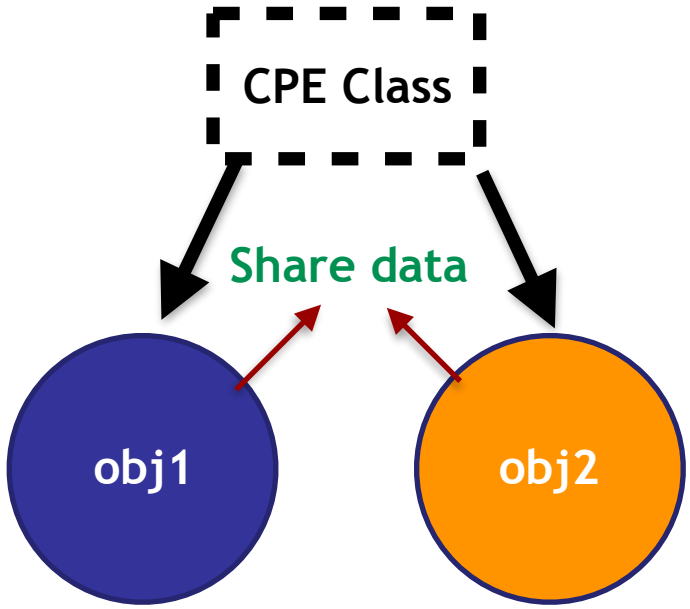
- Modify previous example:

```
class CPE
{
  public:
   int month;
};

int main(){
  CPE obj1;
  CPE obj2;

  obj1.month = 1;
  obj2.month = 2;

  Return 0;
}
```



obj1 != obj2

obj1.month != obj2.month

Static Members

Static member variables

- Place keyword `static` before type
- All objects of class "share" one copy
- Useful for "tracking": e.g., class instance counter

Out-of-class definition (instancing a static variable)

- We also need to declare their definitions outside the class definition.

```
// in Server.h
class Server
{
private:
   static int turn; // this is just a declaration
};

// in Server.cpp
int Server::turn = 0; // now, Server::turn is allocated
int Server::turn = 1; // Not Allowed
```

Static Functions

Member functions can be static

- If no access to object data needed, and still "must" be member of the class.
- Can then be called outside class
  - From non-class objects (e.g., `Server::getTurn();`)
  - As well as via class objects (e.g., `my_server.getTurn();`)

They can use only static data and functions!

- Members that are not static(data, functions) are not accessible inside the static member functions.
  - Because there's no way of knowing from which class instances the non-static members come.

Static Members Example

Display 7.6 Static Members (1 of 4)

```
#include <iostream>
using namespace std;
class Server
{
public:
   Server(char letterName);
   static int getTurn( );
   void serveOne( );
   static bool stillOpen( );
private:
   static int turn;
   static int lastServed;
   static bool nowOpen;
   char name;
};
int Server::turn = 0;
int Server::lastServed = 0;
bool Server::nowOpen = true;
```

Static Members Example

Display 7.6 Static Members (2 of 4)

```
int main( )
{
   Server s1('A'), s2('B');
   int number, count;
   do
   {
       cout << "How many in your group? ";
       cin >> number;
       cout << "Your turns are: ";
       for (count = 0; count < number; count++)
           cout << Server::getTurn( ) << ' ';
       cout << endl;
       s1.serveOne( );
       s2.serveOne( );
   } while (Server::stillOpen( ));
   cout << "Now closing service.\n";
   return 0;
}
```

Static Members Example

Display 7.6 Static Members (3 of 4)

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

//Since getTurn and stillOpen is static, only static members
//can be referenced in here.
int Server::getTurn( ) { turn++; return turn; }

bool Server::stillOpen( ) { return nowOpen; }

void Server::serveOne( )
{
   if (nowOpen && lastServed < turn)
   {
       lastServed++;
       cout << "Server " << name
            << " now serving " << lastServed << endl;
   }
   if (lastServed >= turn) //Everyone served
       nowOpen = false;
}
```

Static Members Example

Display 7.6 Static Members (4 of 4)

```
How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.
```

Static members

```
#include <iostream>
using namespace std;

class Obj {
  public:
    static void printCounter();
    void setName(string str);
    void printInfo();
    static int count;

  private:
    string name;
    static int count2;
};

int Obj::count = 0;
int Obj::count2 = 0;

void Obj::printInfo() {
  cout << name << ": " << count << endl;
}

void Obj::printCounter() {
  //error
  //cout << name << ": " << count << endl;
  cout << count << endl;
}
/*
void setName(string str){
  name = str;
}
*/

void Obj::setName(string str){
  name = str;
}

int main(){
  Obj obj1;
  Obj obj2;
  obj1.setName("obj1");
  obj2.setName("obj2");
  //Obj::setName("test");

  obj1.printCounter();
  obj2.printCounter();
  //Obj::printCounter();

  /*
  Obj::count = 1;

  obj1.printCounter();
  obj2.printCounter();
  */

  //Obj::count2 = 1;
}
```

Separate Interface and Implementation

- **Users of class need not see details of how class is implemented**
  - Principle of OOP → encapsulation
- **Users only need "rules"**
  - Called "interface" for the class
    - In C++: public member functions and associated comments
- **Implementation of class is hidden**
  - Member function definitions elsewhere
  - Users need not see them

String Class

size	Return length of string (public member function)
length	Return length of string (public member function)

```
std::cout << "The size of str is " << str.length() << " bytes.\n";
```

Structures vs. Classes

- **Technically, they are the same**
  - **Unlike C, structures can have member functions!**
  - Perceptionally different mechanisms

- **Structures**
  - Default qualifier is public (private for classes)

- **Classes**
  - Default qualifier is private
  - Typically all data members private
  - Interface member functions public

Thinking Objects

- **Focus for programming changes**
  - Before → algorithms center stage / functions
  - OOP → data is focused / objects
- **Algorithms still exist**
  - They simply focus on their data
  - Are "made" to "fit" the data
- **Designing software solution**
  - Define variety of objects and how they interact