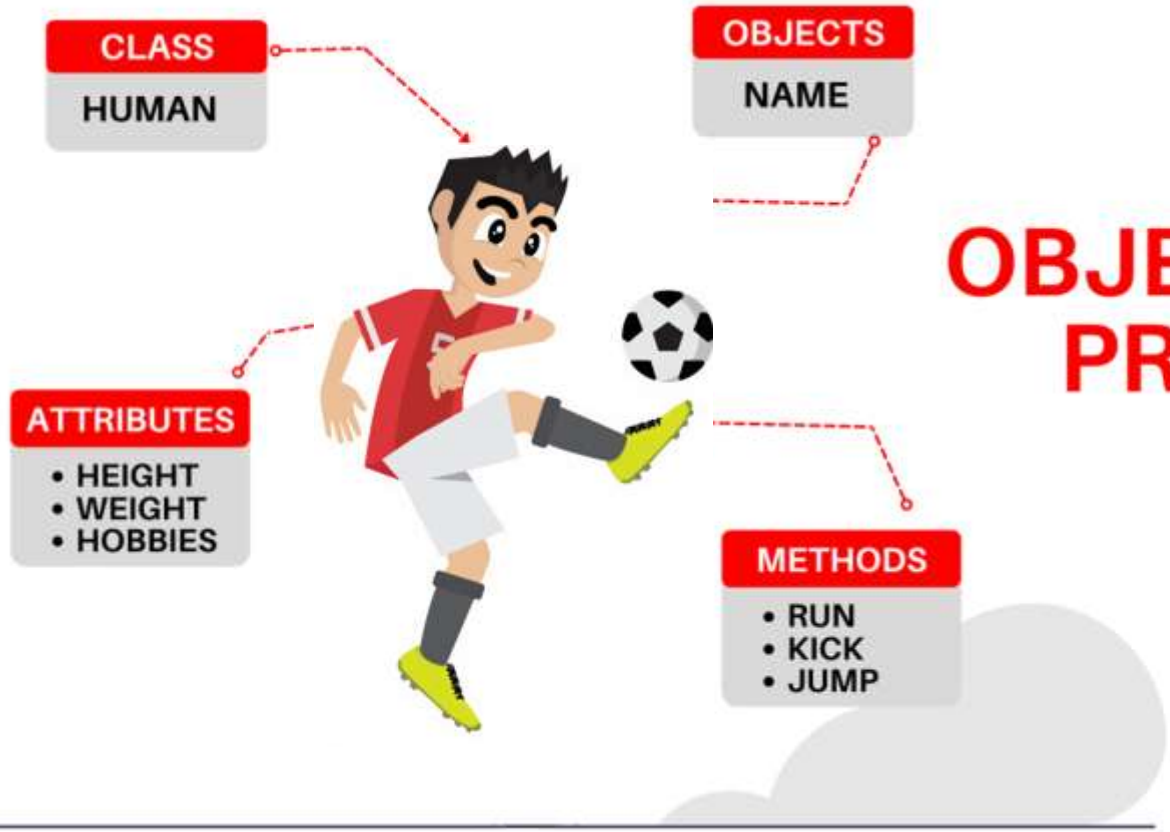
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

Summer 2023

Pir Sami Ullah Shah

Lecture # 6 Classes

Procedural Programming

- Data is stored in a collection of variables and/or structures, along with a set of functions that perform operations on the data

- The data and the functions are separate entities

```
double width;
double length;
```

Data

```
displayWidth();
displayLength();
displayArea();
```

Functions

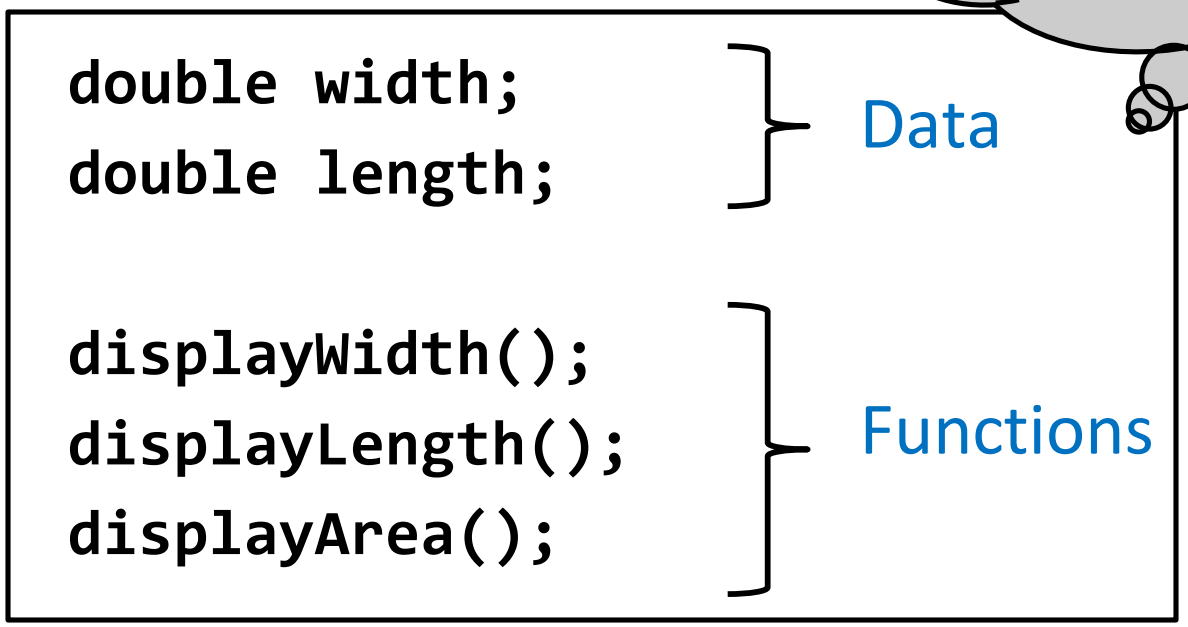
Limitations of Procedural Programming

- Variables and data structures are passed to the functions that perform the desired operations
- What if the data types or data structures change? Many functions must also be changed – Error prone
- As the procedural programs become larger, function hierarchies become more complex:
  - difficult to understand and maintain
  - difficult to modify and extend

Object Oriented Programming

- Procedural programming focuses on creating procedures, object-oriented programming focuses on creating objects
- An object is a self-contained u
data and procedures.

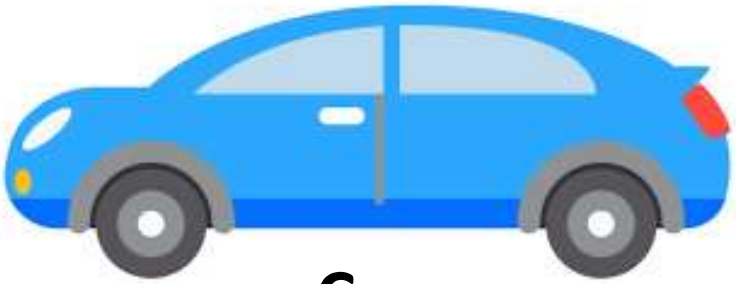
Encapsulation refers to the combining of data and code into a single object



Some Examples of Objects

Light Bulb

- **Attributes**:
  - on (true/false)
- **Behavior**
  - Switch on
  - Switch off
  - Check if on



Car

- **Attributes**:
  - Gas in tank
  - Mileage
- **Behavior**
  - Accelerate
  - Brake
  - Load fuel
  - Check fuel

Bank Account

- **Attributes**:
  - balance
- **Behavior**
  - Deposit
  - Withdraw
  - Check balance

Objects Interact with Each Other

Bank

Get Loan

Customer

Deduct Zakat

Bank Account

Deposit

Withdraw

Object-Oriented Programming Terminology

- **Class**: an abstract data-type or a user-defined data type, similar to a `struct` (allows bundling of related variables)

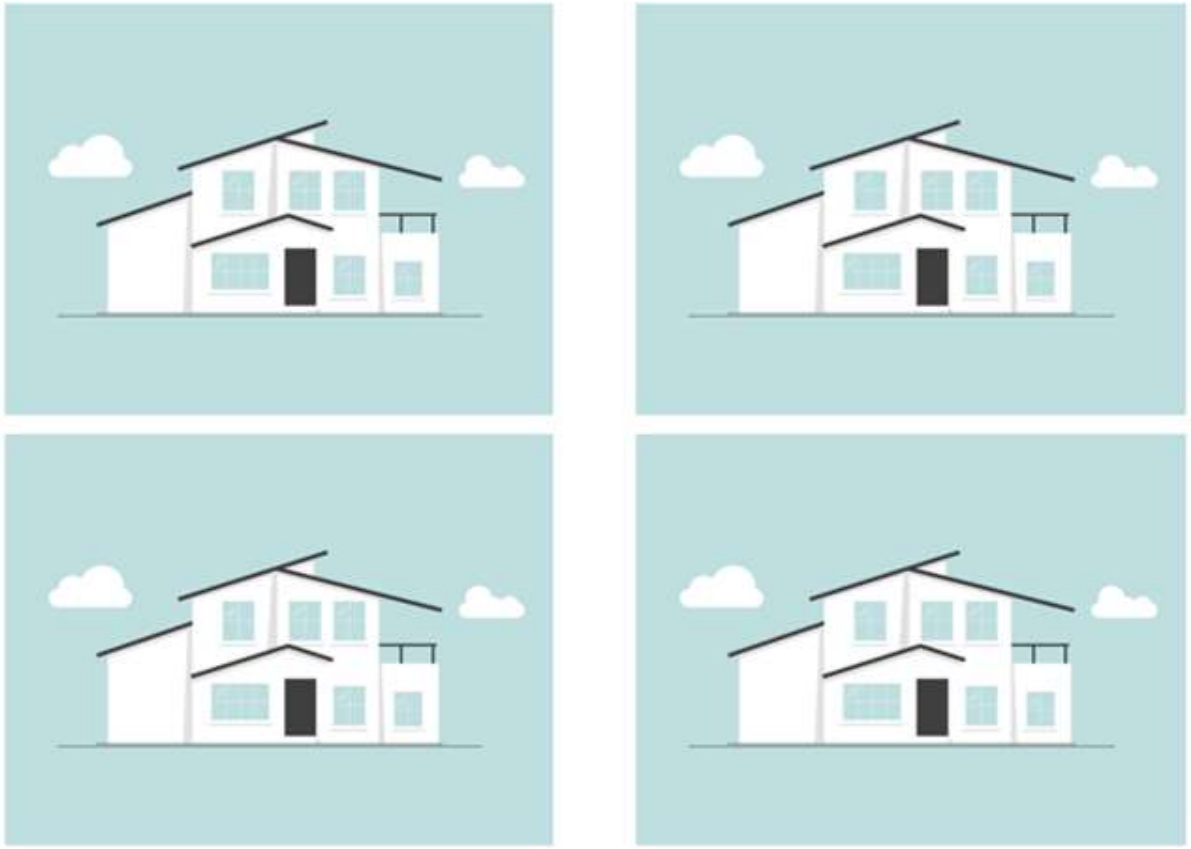
- **Object**: an instance of a `class`, in the same way that a variable can be an instance of a `struct`

Classes and Objects

- A class is like a blueprint and objects are like houses built from the blueprint

Classes and Objects

- An object is an instance (realization) of a class
- A single class can have multiple instances

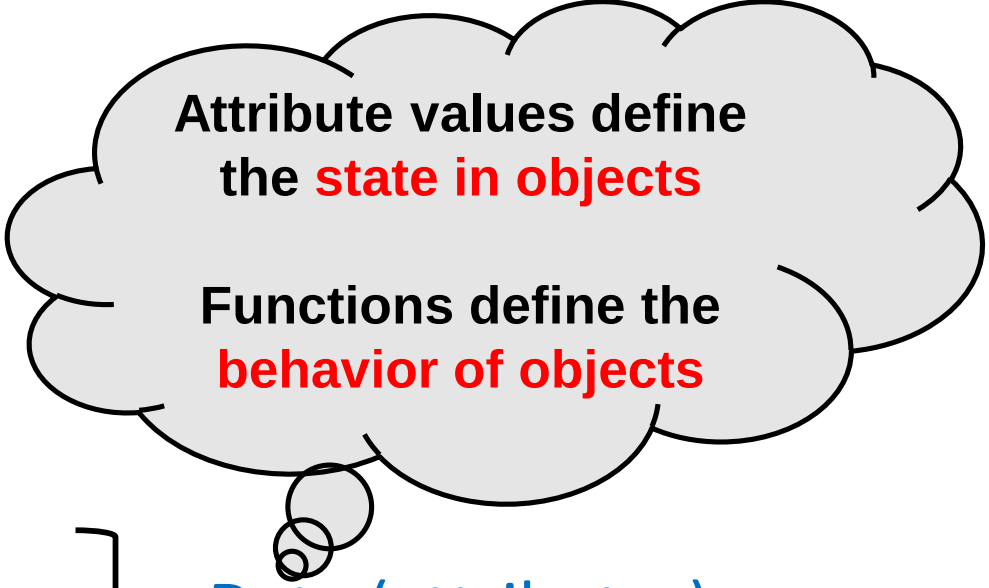


Introduction to Classes

- Objects are created from a **`class`**
- Format:

```
class ClassName
{
        variable declaration;
        methods declaration;
};
```

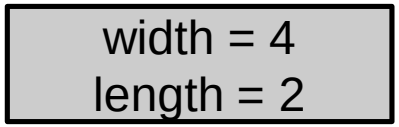
Class Example

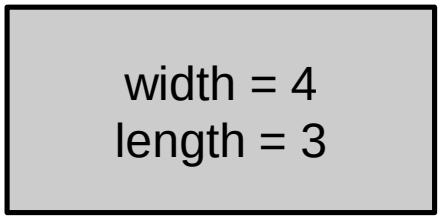


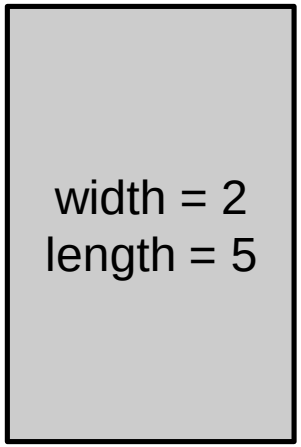

```
class Rectangle
  {
   double width;
   double length;

   displayWidth();
   displayLength();
   displayArea();
  };
```

Data (attributes)

Functions





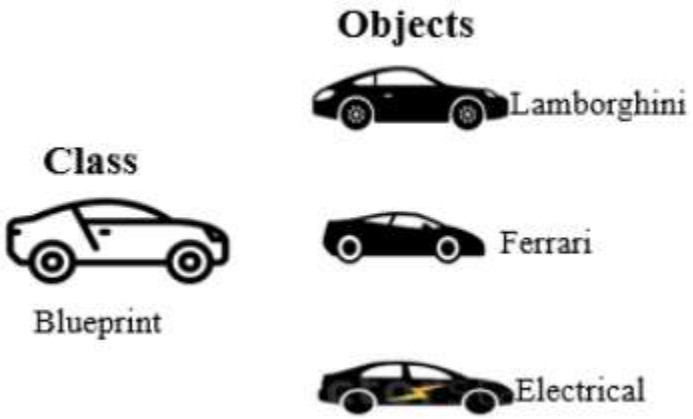


Defining an Instance of a Class

- An object is an instance of a class, to create an object:

```
ClassName objectName;

Rectangle r;
```

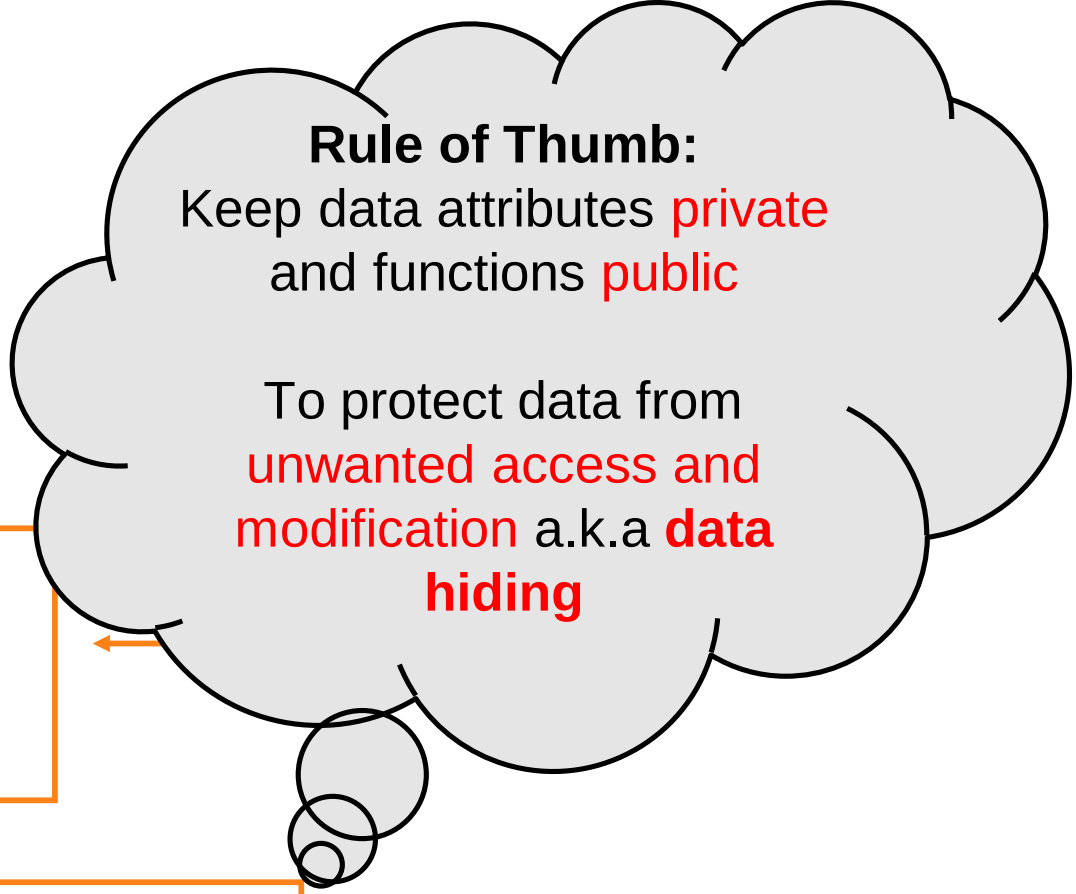
- Every object has a unique identity, its own state



Access Specifiers

- Used to control access to members (attributes and methods) of the class
- **`public:`** can be called / accessed by functions outside of the class
- **`private:`** can only be called / accessed by functions that are members of the class

Class Example




```
class Rectangle
  {
    private:
    double width;
    double length;

    public:
    void displayWidth();
    void displayLength();
    void displayArea();
  };
```

Public Members

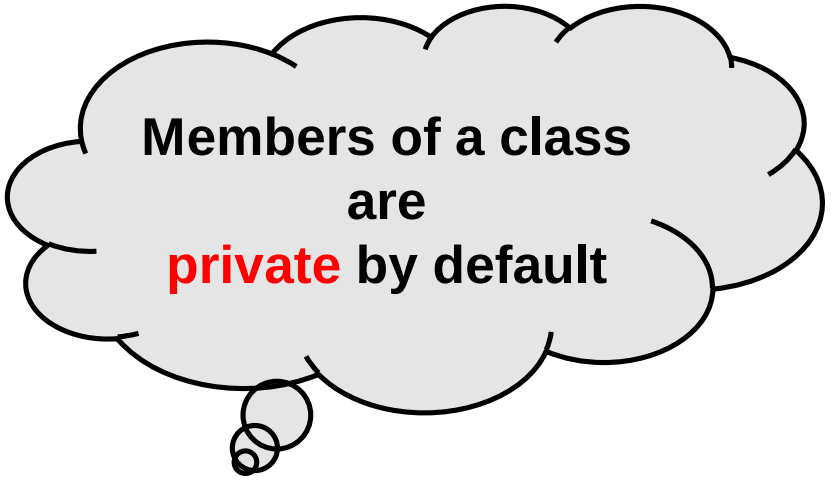
More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is **`private`**

Access Specifiers

- A class is *similar* to struct, but not the same
- Members of a struct are public by default

```
struct Rectangle
  {
   double width;
   double length;
   } r1;
```




```
cout << r1.width; //legal because width is public
```

Access Specifiers

- A class is *similar* to struct, but not the same

- Members of a class are private by default

```
class Rectangle
  {
   double width;
   double length;
  };
Rectangle r1; //class object
cout << r1.width; //ERROR, width is private
```

Access Specifiers

```
class Rectangle {
   private:
      double width;
      double length;
  public:
      double getWidth(){
            return width; }
    };
Rectangle r1; //class object r1
cout << r1.getWidth(); //works getWidth() is
                       public
```

Defining a Member Functions

- Can be defined within the class declaration (in-line member function) or outside the class (out-of-line member functions)

```
class Rectangle {
   private:
      double width;
      double length;
  public:
      double calcArea() {
       return width * length; //inline func }
  };
```

Defining a Member Functions

- When defining a member function outside a class (out-of-line function):
  - Put prototype in class declaration
  - Define function outside using class name and scope resolution operator (`::`)
  - Combine class name with member function name

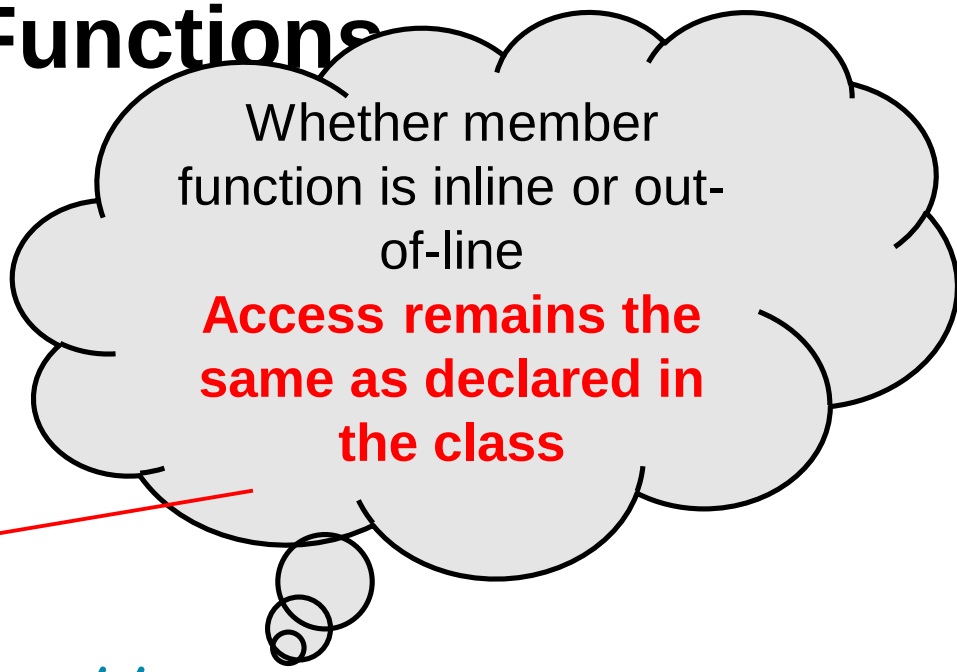
```
returnType ClassName::MemberFunctionName( ){
                    ..........
 }
```

- Different classes can have member functions with the same name

Defining a Member Functions

```
class Rectangle {
   private:
      double width;
      double length;
    public:
      double calcArea(); //prototype
};  //class declaration ends

double Rectangle::calcArea()
{ return width * length; //out-of-line func
}
```

Defining a Member Functions



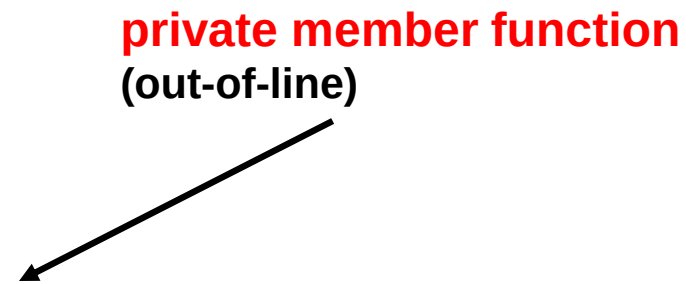

```
class Rectangle {
   private:
      double width;
      double length;
   public:
      double calcArea(); //prototype
};  //class declaration ends

double Rectangle::calcArea()
       {   return width * length;  }
```

Private Member Functions

- Private Member Functions:
  - Only accessible (callable) from member functions of the class
  - No direct access possible (with object instance of the class)
  - Can be: **inline / out-of-line**

Private Member Functions

```
class Rectangle {
   private:
      double width;
      double length;
      double calcArea();

   public:
      void displayArea(){ cout<< calcArea();}
};
double Rectangle::calcArea(){
      return length * width;
}
```



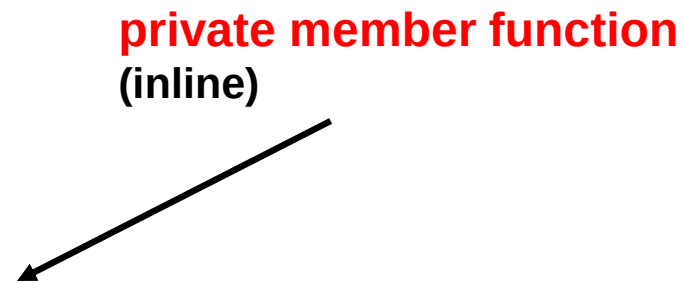
Using Private Member Functions

- A **`private`** member function can only be called by another member function

- It is used for internal processing by the class, not for use outside of the class

Private Member Functions

```
class Rectangle {
   private:
      double width;
      double length;
      double calcArea(){
            return length * width;
      }

   public:
      void displayArea(){ cout<< calcArea();}
};
```



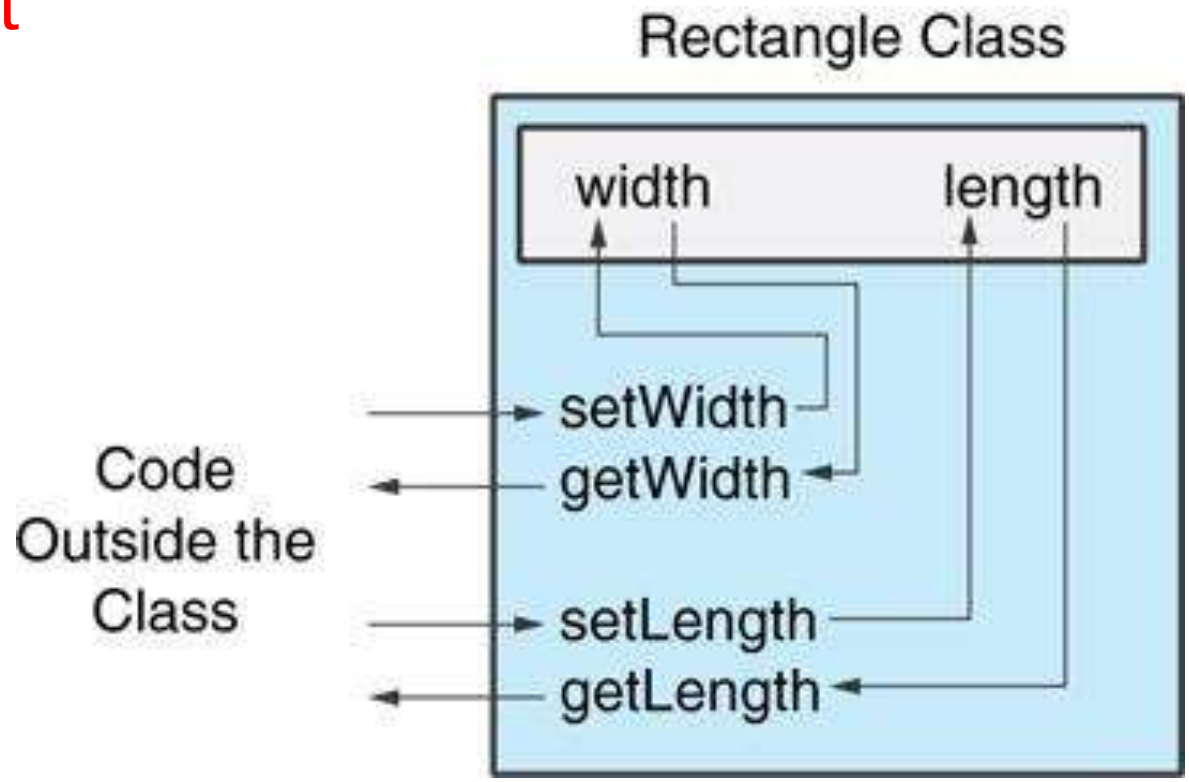
Setters and Getters

- **Setter (Mutator):** a member function that assigns a value to a class data member

- **Getter (Accessor): a** function that reads/gets a value from a class data member.

```
class Rectangle {
   private:
      double width;
      double length;
  public:
      void setWidth(double w){ //setter
            width = w;              }
      void setLength(double l){ //setter
            length = l;             }
      double getWidth(){ //getter
            return width;   }
      double getLength(){ //getter
            return length; }
  }
```

Code outside the class must use the class's public member functions to interact with the object



Constraints can be added in setters to prevent unwanted values in data members

Using `const` With Member Functions

- A **`const`** member function is **read-only**, cannot change the value of any attribute

```
class Rectangle
   {
    private:
    double width;

    public:
    void changeWidth() const {
         width++; //ERROR    }
    };
```

Getters do not change an object's data, so they should be marked `const`.

```
class Rectangle
   {
      private:
         double width;
        double length;
     public:
        void setWidth(double);
        void setLength(double);

        double getWidth() const
           { return width; }

        double getLength() const
           { return length; }

        double getArea() const
           { return width * length; }
  };
```

Avoiding Stale Data

- Some data is the result of a calculation.
- In the `Rectangle` class the area of a rectangle is calculated.
  - length x width
- If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.
- If we change `length` or `width` without updating `area`, then `area` would become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.

Pointer to an Object

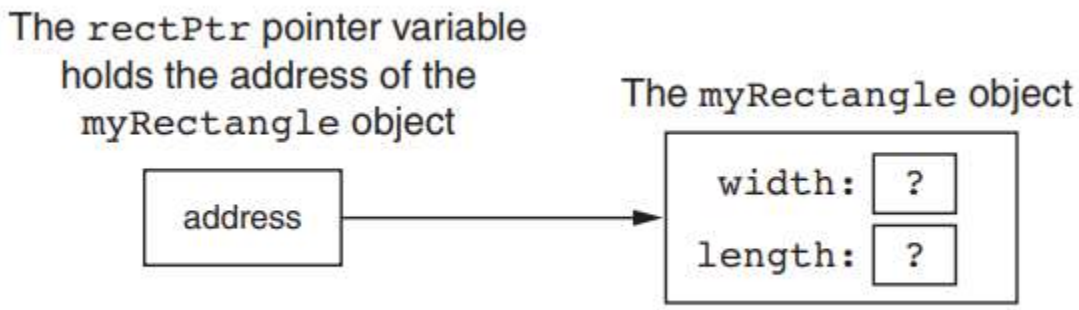
```
Rectangle myRectangle; //Rectangle object
```

- Can define a pointer to an object:

```
Rectangle *rPtr; //Rectangle pointer
```

- Can access public members via pointer:

```
rPtr = &myRectangle;
```



Pointer to an Object

- Can access public members via pointer:

```
rPtr = &myRectangle;
```

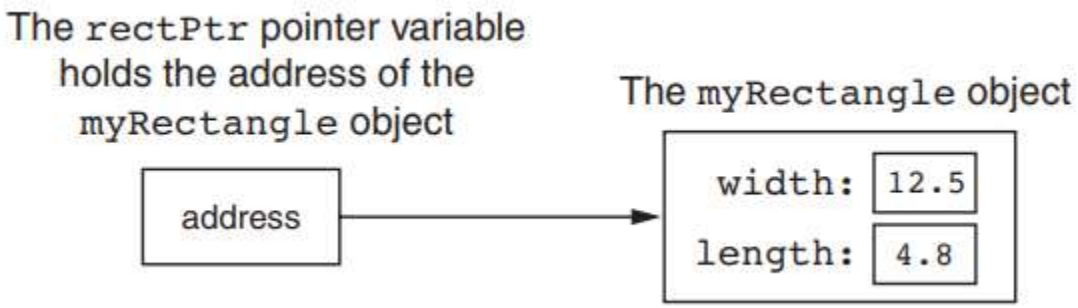
- Recall you can use * and . OR ->

```
rPtr->setWidth(12.5);

cout << rPtr->getLength() << endl;
```

Pointer to an Object

- Can access public members via pointer:

```
rPtr = &myRectangle;
rPtr->setWidth(12.5);
rectPtr->setLength(4.8);
```



Pointer to an Object

- Dynamically allocate objects using a pointer

```
Rectangle *rPtr = new Rectangle;
rPtr->setLength(12.5);
rPtr->setWidth(10.3);
```

- Deallocate memory and delete object

```
delete rPtr;
rPtr = NULL;
```

Reference to Objects

```
// class Count definition
class Count
{
public: // public data is dangerous
   // sets the value of private data member x
   void setX( int value )
   {
      x = value;
   } // end function setX

   // prints the value of private data member x
   void print()
   {
      cout << x << endl;
   } // end function print

private:
   int x;
}; // end class Count
```

```
int main()
{
   Count counter; // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter; // create reference to counter

   cout << "Set x to 1 and print using the object's name: ";
   counter.setX( 1 ); // set data member x to 1
   counter.print(); // call member function print

   cout << "Set x to 2 and print using a reference to an object: ";
   counterRef.setX( 2 ); // set data member x to 2
   counterRef.print(); // call member function print

   cout << "Set x to 3 and print using a pointer to an object: ";
   counterPtr->setX( 3 ); // set data member x to 3
   counterPtr->print(); // call member function print
} // end main
```

- Reference is an alias to an existing object

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Reference and Pointers to Objects

```
class Rectangle
{
   private:
    int w;  int h;

   public:
     Rectangle () {}
    void SetWidth(int ww) { w=ww; }
    void SetHeight(int hh) { h=hh;}
    int getArea() { return w*h; }
};

int main() {
  Rectangle r1;
  Rectangle *ptr = &r1;
  Rectangle &ref = r1;
  Rectangle* &ref2 = ptr;

  r1.SetHeight(5);
  r1.SetWidth(4);
  cout<<"\n Area (object) = "<<r1.getArea();
  cout<<"\n Area (pointer) = "<<ptr->getArea();
  cout<<"\n Area (reference to obj) = "<<ref.getArea();
  cout<<"\n Area (reference to pointer) = "<<ref2->getArea();
  return 0;
}
```

```
Area (object) = 20
Area (pointer) = 20
Area (reference to obj) = 20
Area (reference to pointer) = 20

...Program finished with exit code 0
Press ENTER to exit console.
```

Separating Specification from Implementation

- Makes it easier to modify programs

- Header files (.h)
  - Contains class definitions and function prototypes

- Source-code files (.cpp)
  - Contains member function definitions

```
1   // Fig. 6.5: time1.h
2   // Declaration of the Time class.
3   // Member functions are defined in time1.cpp
4
5   // prevent multiple inclusions o
6   #ifndef TIME1_H
7   #define TIME1_H
8
9   // Time abstract data type definit
10  class Time {
11  public:
12     Time();                          // constructor
13     void setTime( int, int, int ); // set hour,
14     void printMilitary();            // print
15     void printStandard();            // print
16  private:
17     int hour;      // 0 - 23
18     int minute;    // 0 - 59
19     int second;    // 0 - 59
20  };
21
22  #endif
```

Dot (.) replaced with underscore (_) in file name.

If **time1.h** (**TIME1_H**) is not defined (**#ifndef**) then it is loaded (**#define TIME1_H**). If **TIME1_H** *is* already defined, then everything up to **#endif** is ignored.
This prevents loading a header file multiple times.

```
23  // Fig. 6.5: time1.cpp
24  // Member function definitions for Time class.
25  #include <iostream>
26
27  using std::cout;
28
29  #include "time1.h"
30
31  // Time constructor initializes each data member to
32  // Ensures all Time objects start in a consistent
33  Time::Time() { hour = minute = second = 0; }
34
35  // Set a new Time value using military time. Perform
36  // checks on the data values. Set invalid values to
37  void Time::setTime( int h, int m, int s )
38  {
39     hour   = ( h >= 0 && h < 24 ) ? h : 0;
40     minute = ( m >= 0 && m < 60 ) ? m : 0;
41     second = ( s >= 0 && s < 60 ) ? s : 0;
42  }
43
44  // Print Time in military format
45  void Time::printMilitary()
46  {
47     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
48          << ( minute < 10 ? "0" : "" ) << minute;
49  }
50
51  // Print time in standard format
52  void Time::printStandard()
53  {
54     cout << ( ( hour == 0 || hour == 12 ) ? 12 : ho
55          << ":" << ( minute < 10 ? "0" : "" ) <<
56          << ":" << ( second < 10 ? "0" : "" ) <<
57          << ( hour < 12 ? " AM" : " PM" );
58  }
```

Source file uses **#include** to load the header file

Source file contains function definitions

Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type
- A class can have multiple constructors

Constructors

```
class Demo
 {
 public:
 Demo(); // Constructor
 };

 Demo::Demo() //out-of-line
 {
 cout << "Welcome to the constructor!\n";
 }
```

Constructors

```
class Demo
 {
 public:
 Demo(){  //inline function

      // Constructor
       cout << "Welcome to the constructor!\n";
      }
};
```

Constructors

- Whenever an instance of a class is created, the constructor is automatically called

```
Demo dem; //object dem is created



//Prints
Welcome to the constructor!
```

Constructors

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        Rectangle();                    // Constructor
        void setWidth(double);
        void setLength(double);

        double getWidth() const
            { return width; }

        double getLength() const
            { return length; }

        double getArea() const
            { return width * length; }
};
```

```
Rectangle::Rectangle()
{
    width = 0.0;
    length = 0.0;
}
```

What Constructors Do

- Help in initializing class data members

```
Employee( ) { id = 0; }
```

- Allocate memory for dynamic members

```
Employee() {
char* nameptr = new char[20];
}
```

- Allocate any needed resources
  - Such as to open files, etc.

Default Constructors

A class can have **only one** default constructor

- A default constructor is a constructor that takes no arguments

- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.

- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```

Parametrized Constructors

- To create a constructor that takes arguments:
  - indicate parameters in prototype:

```
Rectangle(double, double);
```

  - Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len)
{
    width = w;
    length = len;
}
```

Parametrized Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10.7, 5.2);
```

- Constructors can be overloaded

```
Rectangle(double w, double len, double ar)
{
   width = w;
   length = len;
   area = ar;
}
Rectangle r(10.0, 5.0, 50.0);
```

More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle(double len = 0, double wid = 0){
      length = len;  width = wid;
}
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

- Cannot create multiple default constructors - ERROR

More About Default Constructors

```
Rectangle(double len = 0, double wid = 0){
      length = len;  width = wid;
}
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
Rectangle r1(10.0);
Rectangle r2(10.0, 2.0);
```

All three cause the same constructor to execute

More About Default Constructors

```
Rectangle(double len = 0, double wid = 0){
      length = len;  width = wid;
}
```

- Creating an object and pas
  this constructor to execu

```
Rectangle r;
Rectangle r1(10.0);
Rectangle r2(10.0, 2.0);
```

With this constructor, cannot create any other constructor with 0,1 or 2 parameters.
Because then the compiler cannot differentiate and decide which one to call

All three cause the same constructor to execute

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.

Destructors

- Member function automatically called when an object is destroyed

- Destructor name is ~classname, *e.g.*, **`~Rectangle`**
- Has no return type; takes no arguments

- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

Example InventoryItem.h

Contents of InventoryItem.h (Version 1)

```
1  // Specification file for the InventoryItem class.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <cstring>   // Needed for strlen and strcpy
5
6  // InventoryItem class declaration.
7  class InventoryItem
8  {
9  private:
10    char *description;  // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
```

Example `InventoryItem.h`

```
13  public:
14     // Constructor
15     InventoryItem(char *desc, double c, int u)
16        { // Allocate just enough memory for the description.
17          description = new char [strlen(desc) + 1];
18
19          // Copy the description to the allocated memory.
20          strcpy(description, desc);
21
22          // Assign values to cost and units.
23          cost = c;
24          units = u;}
25
26     // Destructor
27     ~InventoryItem()
28        { delete [] description; }
29
```

(continued)

Constructors, Destructors, and Dynamically Allocated Objects

- When an object is dynamically allocated with the new operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```

Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

Member Function Overloading

- Non-constructor member functions can also be overloaded:

```
void setCost(double);
void setCost(char *);
```

- Must have unique parameter lists, like overloaded constructors

When Constructors and Destructors Are Called

Constructors and destructors called automatically

1.Global scope objects

- Constructors called before any other function (including main)
- Destructors called when main terminates (or exit function called)

1.Local scope objects

- Constructors called when objects are defined
- Destructors called when objects leave scope

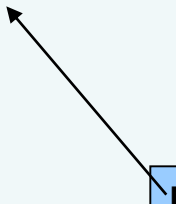
```
7       class CreateAndDestroy {
8       public:
9          CreateAndDestroy( int );  // constructor
10         ~CreateAndDestroy();      // destructor
11      private:
12         int data;
13      };
14
15      #endif


24
25      CreateAndDestroy::CreateAndDestroy( int value )
26      {
27         data = value;
28         cout << "Object " << data << "
29      }
30
31      CreateAndDestroy::~CreateAndDestroy()
32         { cout << "Object " << data << "   destructor
" << endl; }
```

Constructor and Destructor changed to print when they are called.

```
42
43      void create( void );
44
45      CreateAndDestroy first( 1 );  // global object
46
47      int main()
48      {
49         cout << "   (global created before main)" <<
50
51         CreateAndDestroy second( 2 );        // local
52         cout << "   (local in main)" << endl;
53
54
55
56
57         create();  // call function to create objects
58
59         CreateAndDestroy fourth( 4 );        // local
60         cout << "   (local in main)" << endl;
61         return 0;
62      }
```

```
63
64      // Function to create objects
65      void create( void )
66      {
67         CreateAndDestroy fifth( 5 );
68         cout << "   (local in create)" <<
69
70
71
72
73         CreateAndDestroy seventh( 7 );
74         cout << "   (local in create)" <<
75      }
```

```
OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local in main)
Object 5   constructor   (local in create)
Object 7   constructor   (local in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local in main)
Object 4   destructor
Object 2   destructor
Object 1   destructor
```



Notice how the order of the constructor and destructor call depends on the types of variables (local and global) they are associated with.

Destructor Example

```
void f1()
{
   Employee *c = new Employee[3];
   c[0].var1 = 322;
   c[1].var1 = 5
   c[2].var1 = 9;

}
```

```
class Item {
private:
      double cost;
      int units;
public:
Item() {                    //Default Constructor
      cost = 0.0;
      units = 0; }

Item(int costVal){     //Constr 1 parameter
      cost = costVal;
      units = 0; }

Item(double c, int u) { //Constr 2 parameters
      cost = c;
      units = u; } };
```

Arrays of Objects

- Objects can be the elements of an array:

```
Item inventory[40];
```

- Default constructor for object is used when array is defined

Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

```
Item inventory[3] =
  { 22.4, 10.30, 99.0 };
```

- The compiler treats each item in the initializer list as an argument for an array element's constructor

- Second constructor in the Item class

Arrays of Objects

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
Item inventory[] = {Item(6.95, 12),
                    Item(8.75, 20),
                    Item(3.75, 10) };
```

Arrays of Objects

- It isn't necessary to call the same constructor for each object in an array:

```
Item inventory[] = {20.5,
                    Item( 8.75, 20), 45.0};
```

Accessing Objects in an Array

- Objects in an array are referenced using subscripts

- Member functions are referenced using dot notation:

```
inventory[2].setUnits(30);

cout << inventory[2].getUnits();
```

Array of pointers to objects

- Declare an array of pointer to objects
- Allocate and initialize each object in a loop

```
Date *dates[31];

for (int day = 0; day < 31; ++day)
{
  dates[day] = new Date(3, day, 2020);
}
```

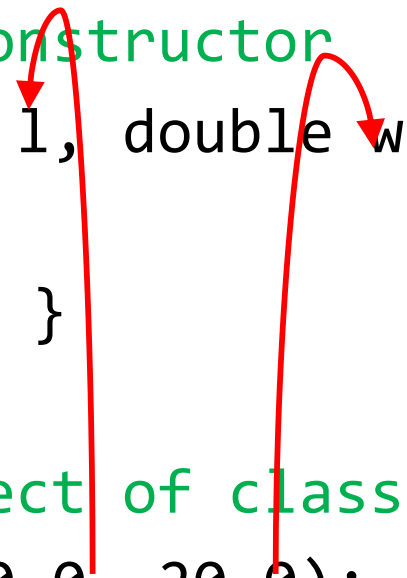
Default Member-wise Assignment

- Assignment operator (**=**) can be used to assign an object to another object of the **same type**.

- Member-wise assignment: each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left

Default Member-wise Assignment

```
class Rectangle{
      double length;
      double width;
public:
Rectangle(){ //default constructor
      length = 0; width = 0;    }
//parametrized constructor
Rectangle(double l, double w) {
      length = l;
      width = w;   }
};
//box1 is an object of class Rectangle
Rectangle box1(10.0, 20.0);
```

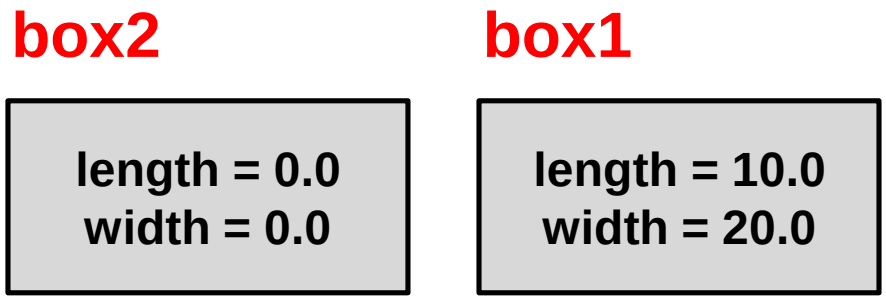


box1

length = 10.0
width = 20.0

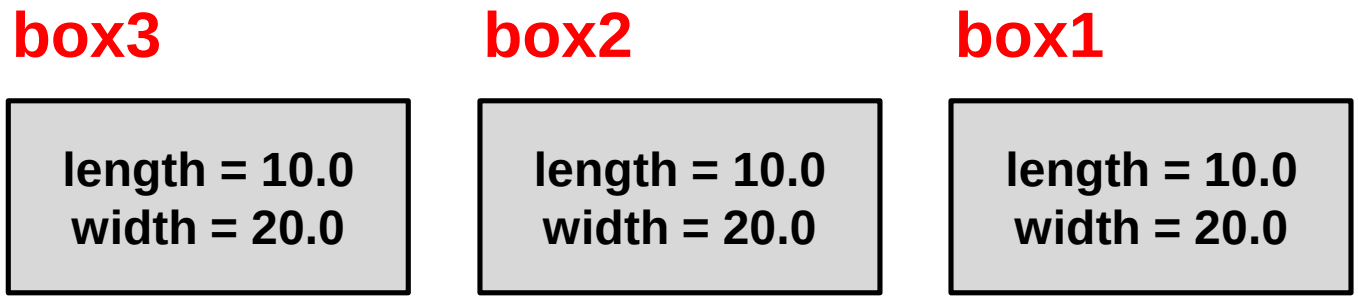
Default Member-wise Assignment

```
//box1 is an object of class Rectangle
Rectangle box1(10.0, 20.0);

Rectangle box2;

box2 = box1;
```



Default Member-wise Assignment

```
//box1 is an object of class Rectangle
Rectangle box1(10.0, 20.0);

Rectangle box2;

box2 = box1;

Rectangle box3 = box2;
```



```
class Date
{
public:
   Date( int = 1, int = 1, int = 2000 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
}; // end class Date

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

```
int main()
{
   Date date1( 7, 4, 2004 );
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1; // default memberwise assignment

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;
} // end main
```

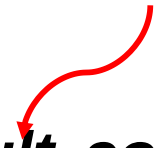
```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Default Copy Constructor

- A ***copy constructor*** creates an object and initializes it with another object's data. Objects must be of the same type

- If a class doesn't have a copy constructor, C++ creates a default copy constructor for it.

```
Rectangle box3 = box2; OR Rectangle box3(box2);
```



- The ***default copy constructor*** performs the member-wise assignment when an object is initialized using another object.

Copy constructor

- A type of constructor that is used to initialize an object with another object of the same type is known as copy constructor.
- It is by default available in all classes
- It has the same form as other constructors, except it has a reference parameter of the same class type
- syntax is **`ClassName(ClassName &Variable)`**

  `Rectangle(Rectangle &r)`

Copy Constructor for Class Date

```
Date::Date(Date &date)
{
   month = date.month;
   day   = date.day;
   year  = date.year;
}
```

Uses of the Copy Constructor

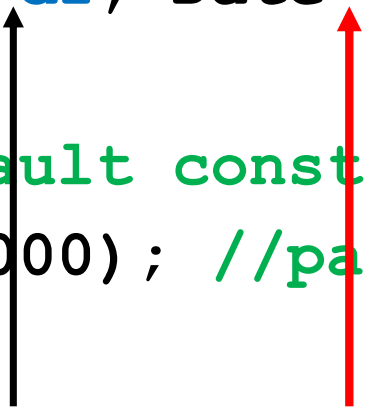
- Implicitly called in 3 situations:
  1. Dynamically defining a new object from an existing object
  2. passing an object by value
  3. returning an object by value

Copy Constructor: Defining a New Object

```
//parametrized constructor called
Date d1(2,28,2020);

// initialize 2 local objects from d1
Date d2(d1);// pass by value
Date d3 = d1;// return value

// init a dynamic object from d1
Date* pdate = new Date(d1);
```

Copy Constructor: Passing Objects by Value

```
//copy constructor called for each value arg
int dateDiff(Date d1, Date d2);
...
Date today; //default constructor called
Date d3(02, 21, 2000); //parametrized constr.

cout << dateDiff(d3, today);
```

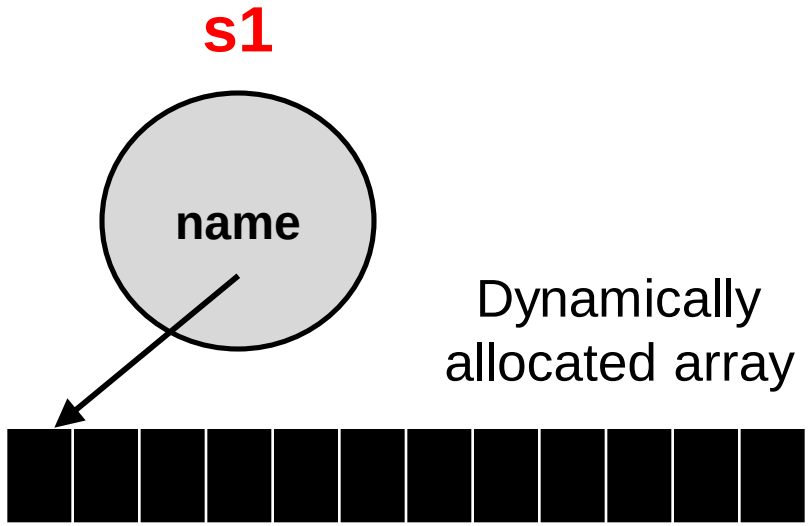


Some Issues with the Default Copy Constructor

- There are instances, where the default copy constructor can be problematic

```
class Student{
private:
      char *name;
public:
Student(){ //default constructor
      name = new char[12]{0};
}

void setName(char* nameVal){  //setter for name
      int i=0;
      while((*nameVal)!='\0'){
            name[i++]=(*nameVal);
            nameVal++;    }
}
char* getName(){ //getter for name
      return name; }    };
```

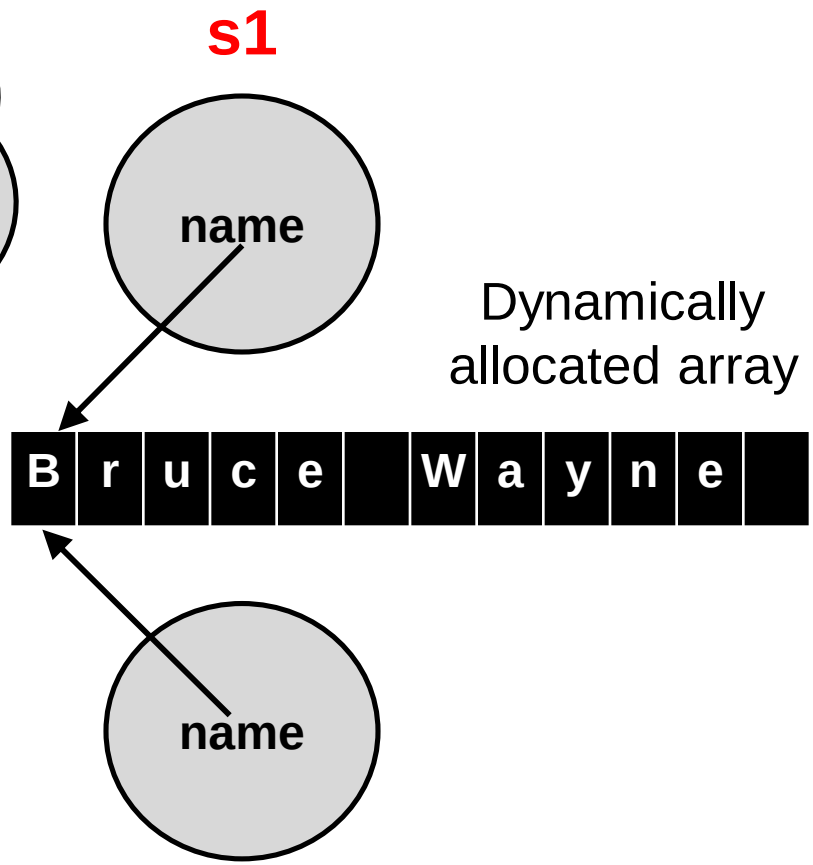
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
```




```
return 0; }
```

```
int main
Studen
char
s1.set
cout<<s
Student s2 = s1;
return 0; }
```

s2's **default copy constructor is called** and s1's member variable values are **copied in s2**

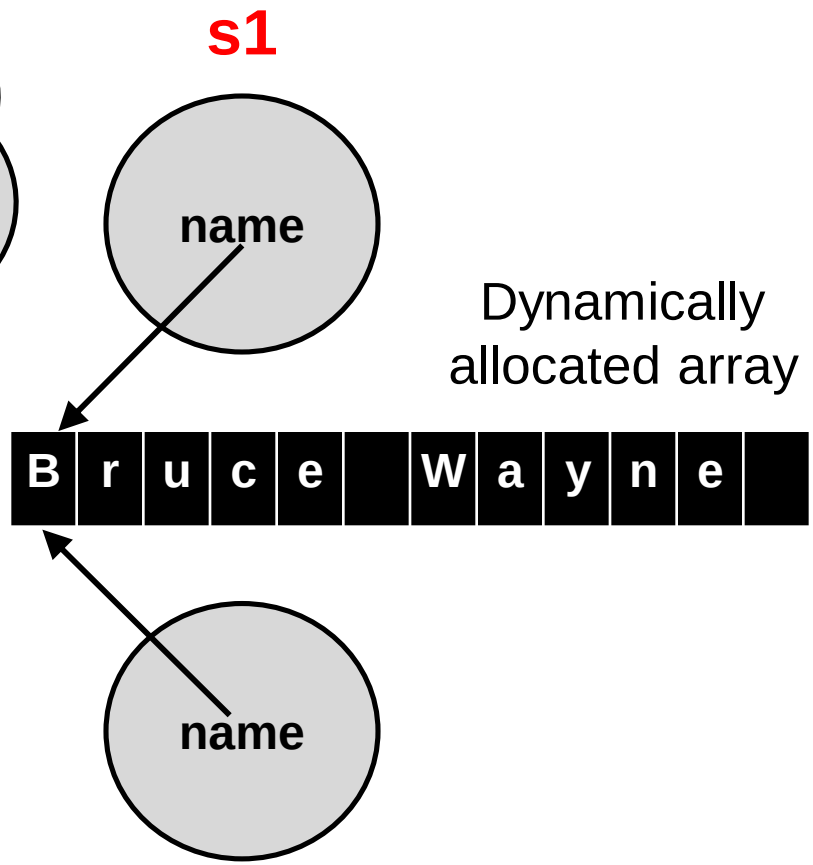


Output

Bruce Wayne

```
int main
Studen
char
s1.set
cout<<s
Student s2 = s1;
cout<<s2.getName()<<endl;
return 0; }
```

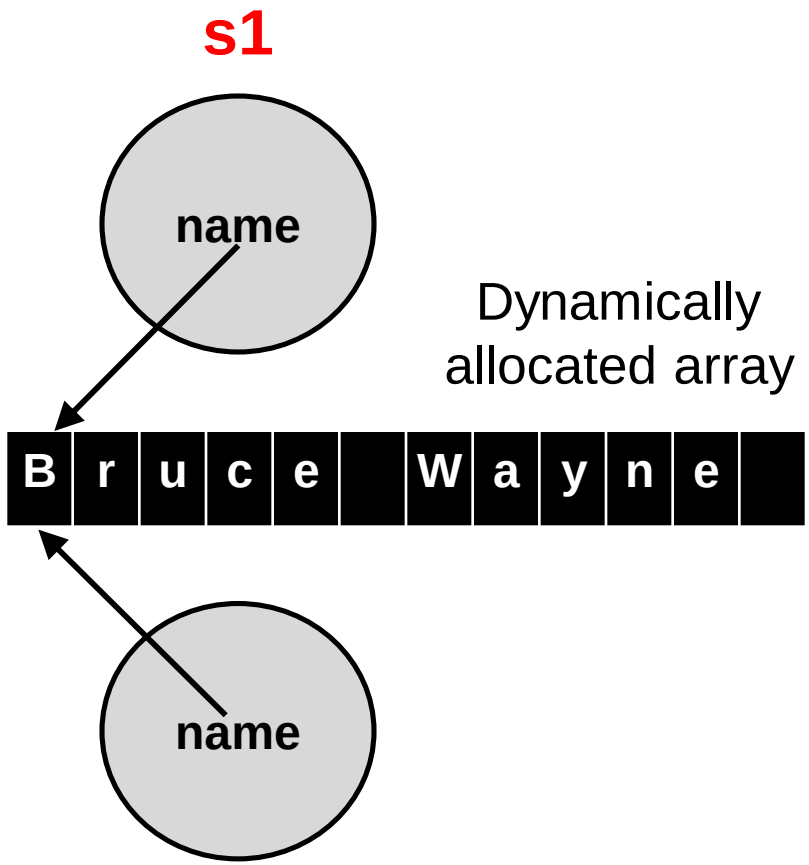
Pointers in both objects will point to the same address a.k.a *shallow copy*



Output

Bruce Wayne

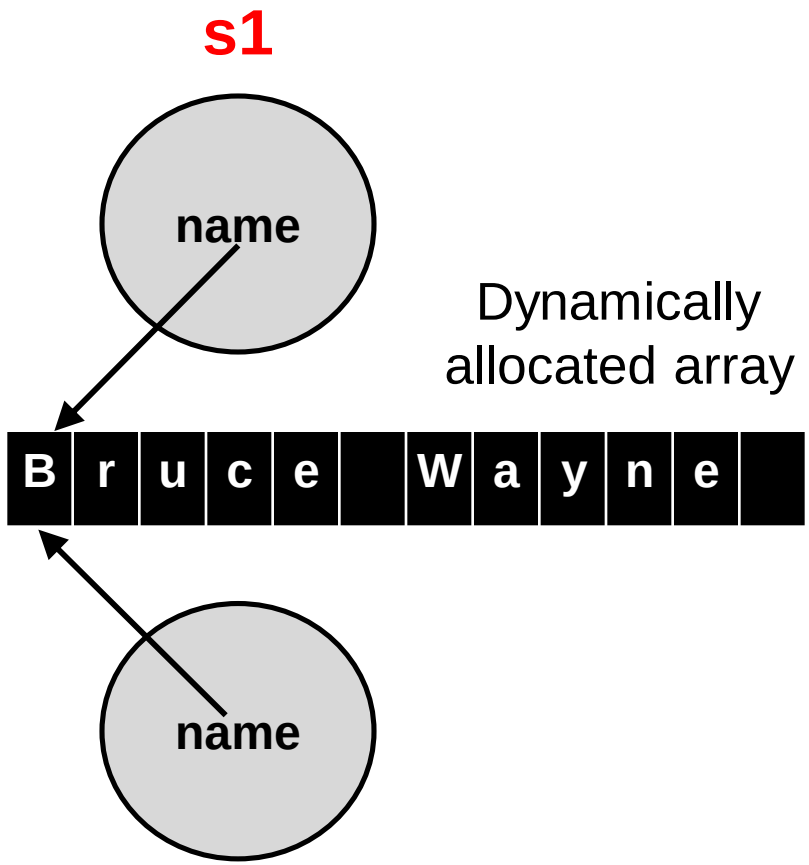
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

return 0; }
```



Output

Bruce Wayne
Bruce Wayne

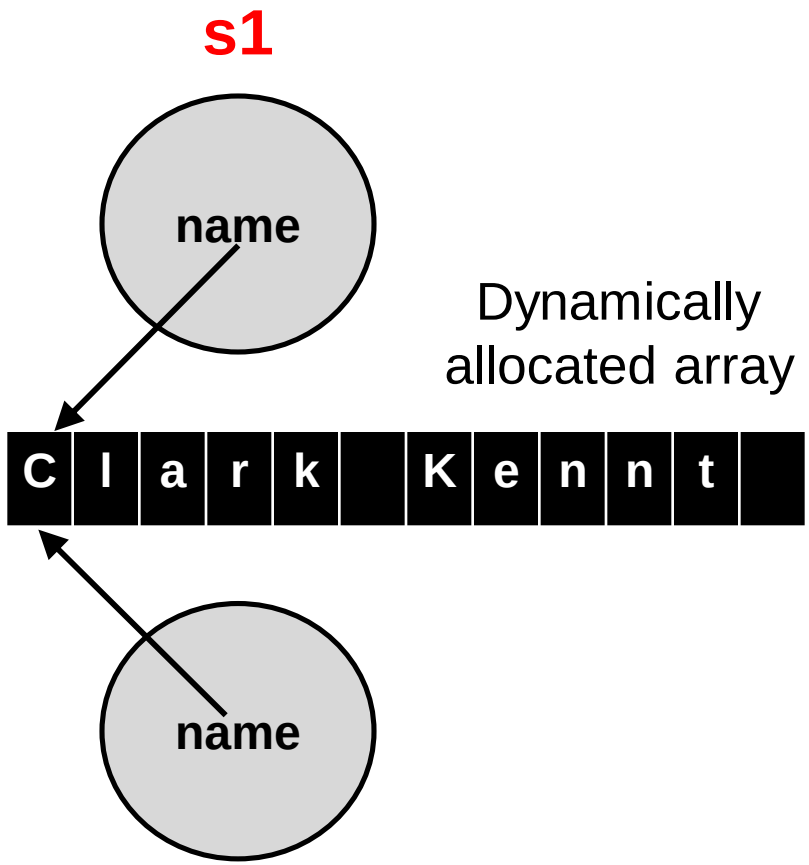
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);



return 0; }
```



Output

Bruce Wayne
Bruce Wayne

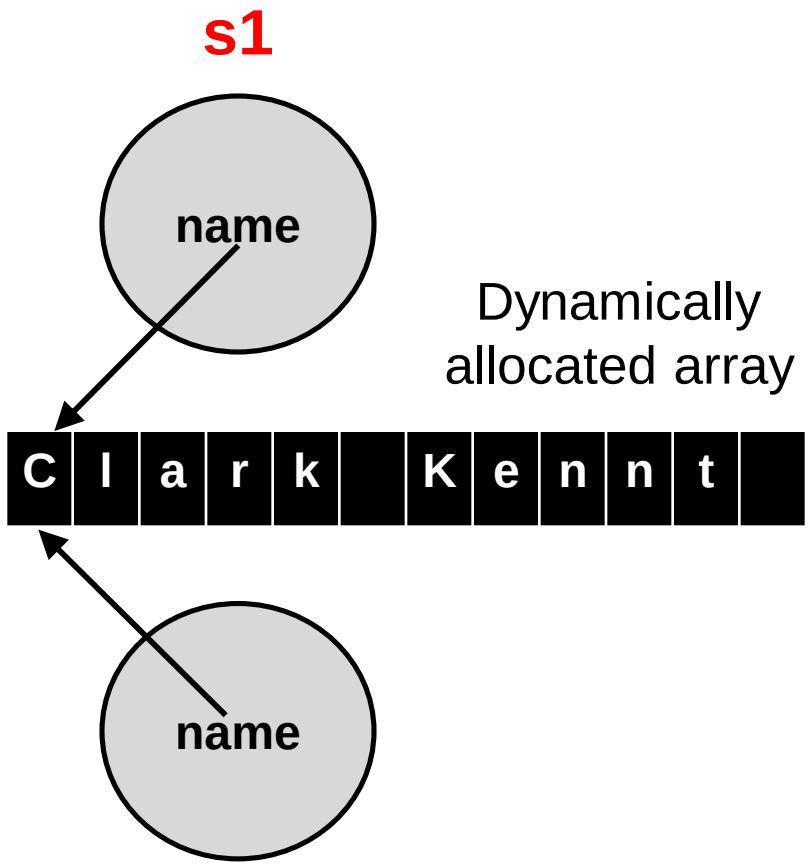
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);
cout<<s2.getName()<<endl;


return 0; }
```



Output

Bruce Wayne
Bruce Wayne

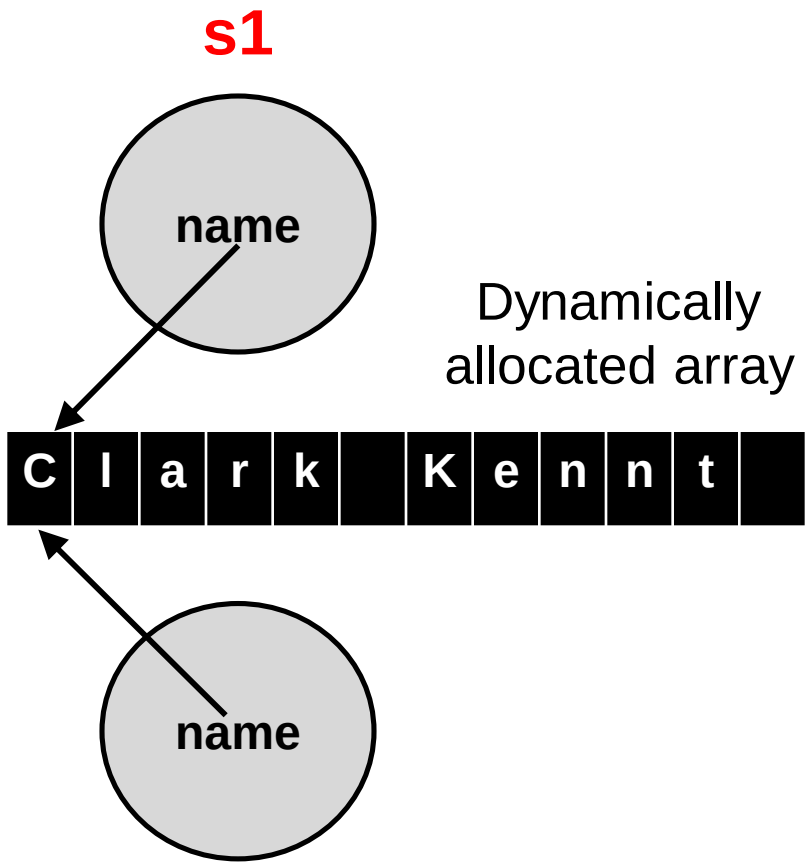
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);
cout<<s2.getName()<<endl;



return 0; }
```



Output

Bruce Wayne
Bruce Wayne
Clark Kennt

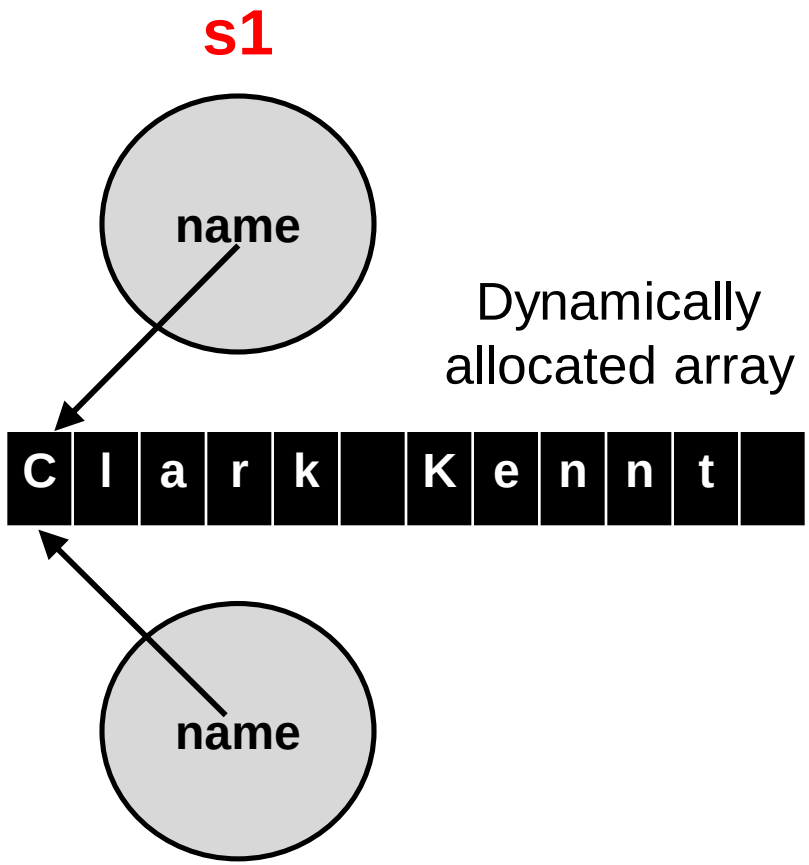
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);
cout<<s2.getName()<<endl;
cout<<s1.getName()<<endl;

return 0; }
```



Output

Bruce Wayne
Bruce Wayne
Clark Kennt

```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);
cout<<s2.getName()<<endl;
cout<<s1.getName()<<endl;

return 0; }
```



Output

Bruce Wayne
Bruce Wayne
Clark Kennt
Clark Kennt

Some Issues with Default Copy Constructor – *Shallow Copy*

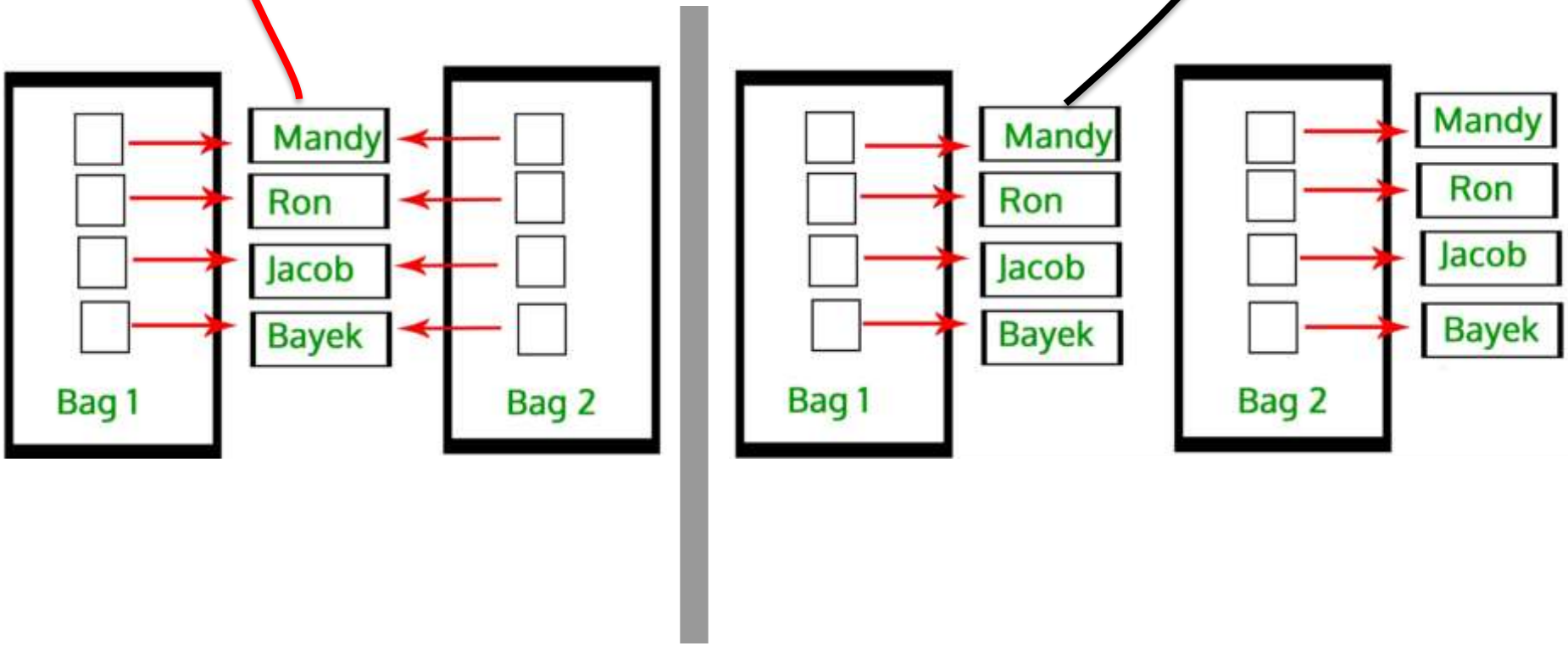
- Either object can manipulate the values stored in the array, causing the changes to show up in the other object.

- One object can be destroyed, causing its destructor to be called, which frees the allocated memory

- The remaining object's *name* pointer would still reference this section of memory, although it should no longer be used

User-defined Copy Constructor, required?

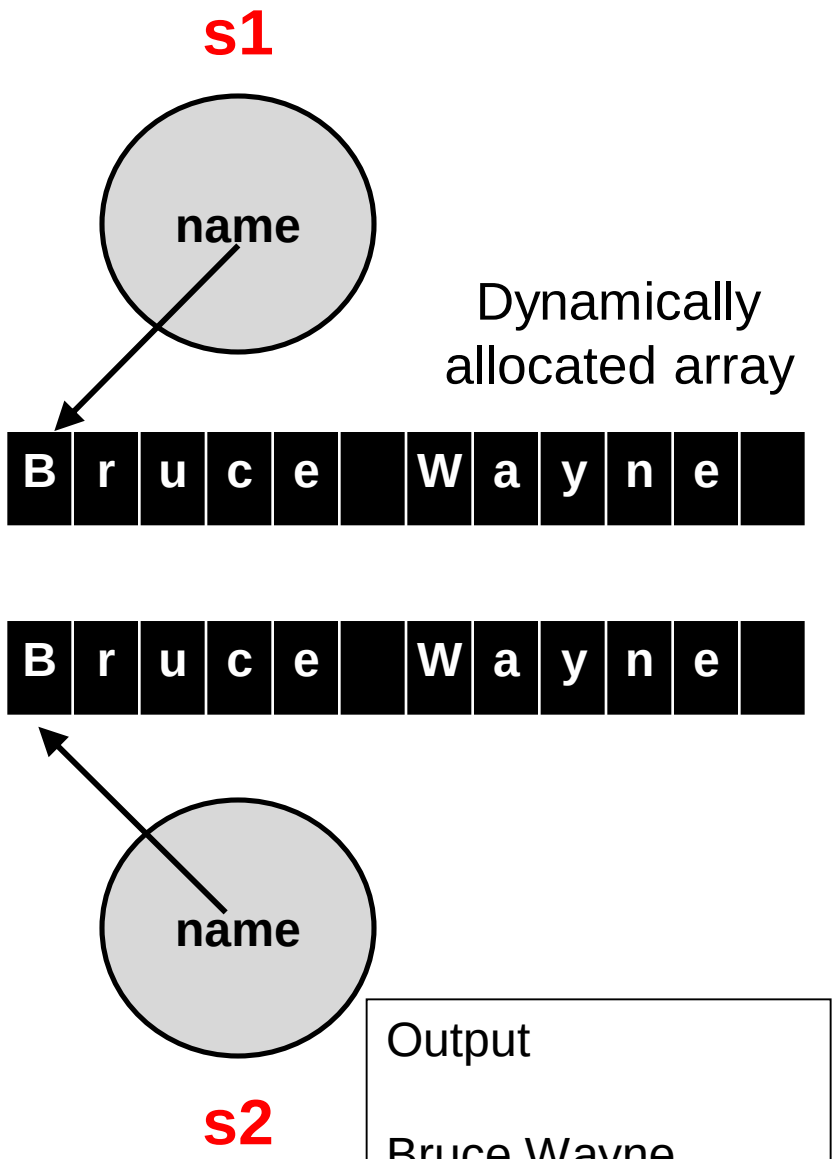
- Default-copy Constructor do only “Shallow Copy”
- We need user-defined copy-constructor,
  - When we need “Deep Copy” (for Dynamic Memory)



User defined Copy Constructor – *Deep Copy*

```
class Student{
private:
      char *name;
public:
Student(){ //default constructor
      name = new char[12]{0};
      }
Student(const Student &s){ //copy constructor
      name = new char[12]; //for deep copy
            for (int i = 0; i < 12; i++) {
            name[i] = s.name[i];   }
      }
};
```

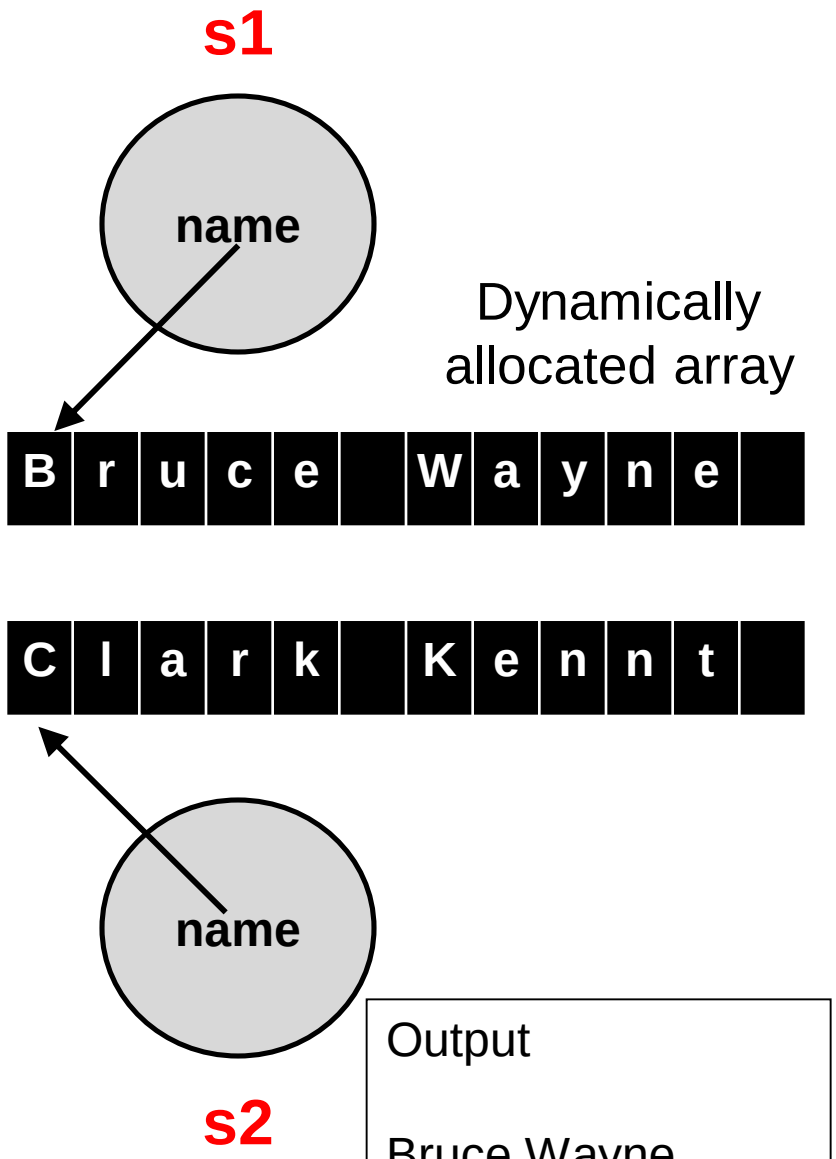
```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);



return 0; }
```



Output

Bruce Wayne
Bruce Wayne

```
int main(){
Student s1;
char name1[]="Bruce Wayne";
s1.setName(name1);
cout<<s1.getName()<<endl;

Student s2 = s1;
cout<<s2.getName()<<endl;

char name2[]="Clark Kennt";
s2.setName(name2);
cout<<s2.getName()<<endl;
cout<<s1.getName()<<endl;

return 0; }
```

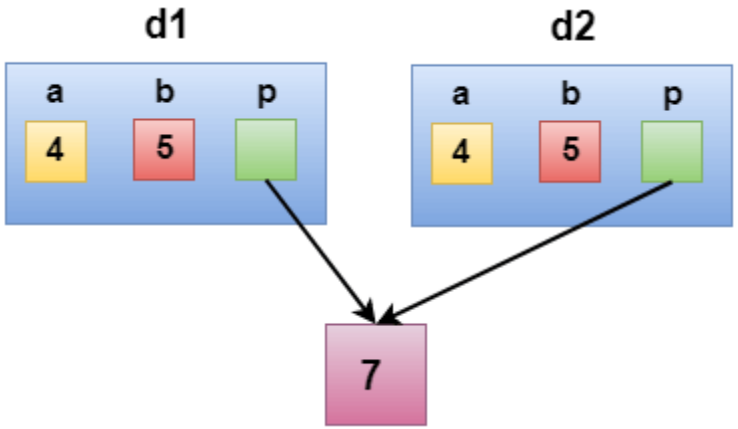


Output

Bruce Wayne
Bruce Wayne
Clark Kennt
Bruce Wayne

Shallow Copy

```
1.  class Demo
2.  {
3.     int a;
4.     int b;
5.     int *p;
6.     public:
7.     Demo()
8.     {
9.        p=new int;
10.    }
11.    void setdata(int x,int y,int z)
12.    {
13.       a=x;
14.       b=y;
15.       *p=z;
16.    }
17.    void showdata()
18.    {
19.       std::cout << "value of a is : " <<a<< std::endl;
20.       std::cout << "value of b is : " <<b<< std::endl;
21.       std::cout << "value of *p is : " <<*p<< std::endl;
22.    }
23. };
```



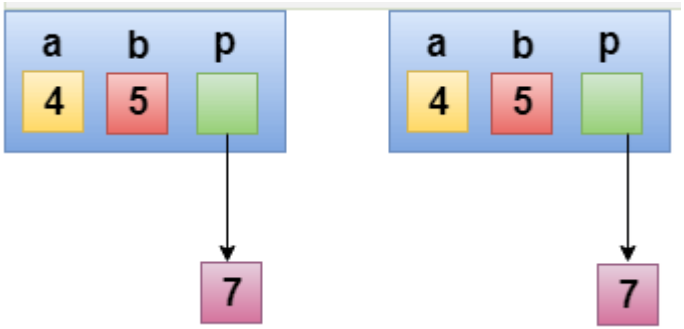

```
24. int main()
25. {
26.   Demo d1;
27.   d1.setdata(4,5,7);
28.   Demo d2 = d1;
29.   d2.showdata();
30.   return 0;
31. }
```

Deep Copy

```
1.  class Demo
2.  {
3.      public:
4.      int a;
5.      int b;
6.      int *p;
7.
8.      Demo()
9.      {
10.         p=new int;
11.     }
12.     Demo(Demo &d)
13.     {
14.         a = d.a;
15.         b = d.b;
16.         p = new int;
17.         *p = *(d.p);
18.     }
19.     void setdata(int x,int y,int z)
20.     {
21.         a=x;
22.         b=y;
23.         *p=z;
24.     }
```

Deep Copy

```
1.
2.      void showdata()
3.      {
4.          std::cout << "value of a is : " <<a<< std::endl;
5.          std::cout << "value of b is : " <<b<< std::endl;
6.          std::cout << "value of *p is : " <<*p<< std::endl;
7.      }
8.  };
9.  int main()
10.{
11.   Demo d1;
12.   d1.setdata(4,5,7);
13.   Demo d2 = d1;
14.   d2.showdata();
15.   return 0;
16.}
```



Copy Constructor for Class Date

- Copy constructors have to use reference parameters so they have access to their argument's data.

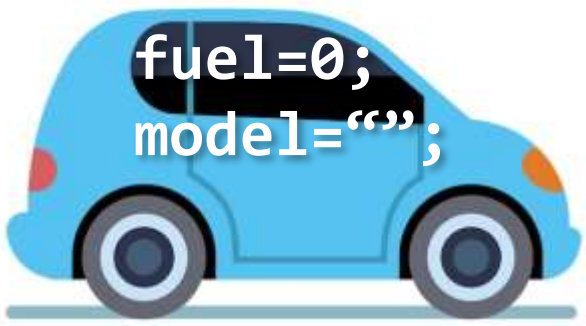
- To prevent the copy constructor from modifying the arguments data, make the copy constructors' parameters ***constant***

```
Date::Date(const Date &date)
{ month = date.month;
  day   = date.day;
  year  = date.year;
}
```

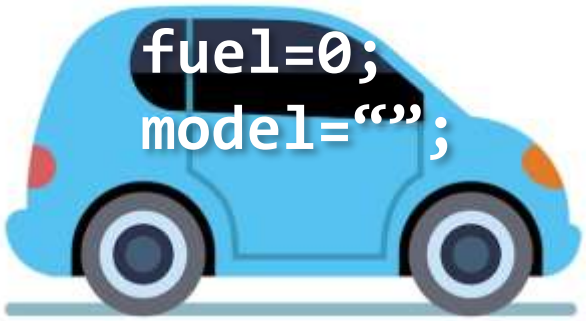
Class Instance Variables (Attributes)

```
class Car {
      int fuel;
      string model;
public:
Car() { //constructor
      fuel = 0;
      model = “”;}
};
```

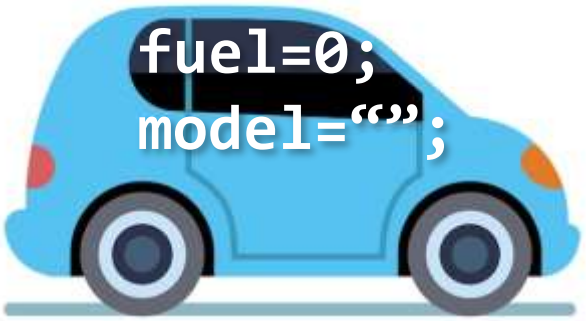
```
Car car1, car2, car3;
```



car1



car2

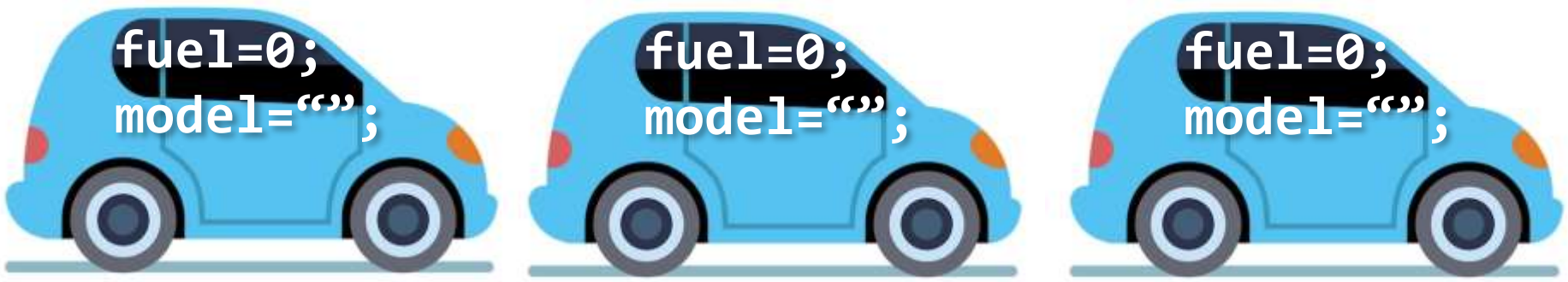


car3

Class Instance Variables (Attributes)

```
class Car {
      int fuel;
      string model;
public:
Car() { //constructor
      fuel = 0;
      model = “”;}
};
```

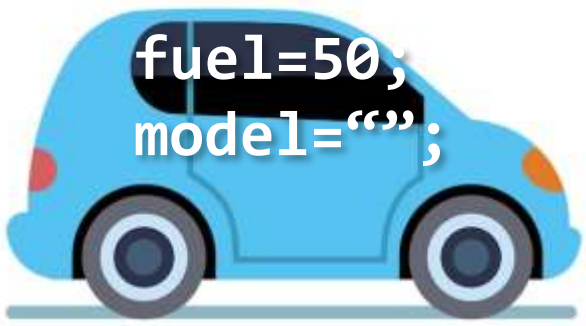
```
Car car1, car2, car3;

//assume class has a
setter
car1.setFuel(50);
```



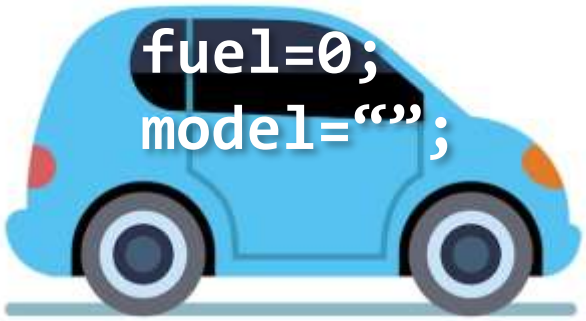
Class Instance Variables (Attributes)

```
class Car {
      int fuel;
      string model;
public:
Car() { //constructor
      fuel = 0;
      model = “”;}
};
```

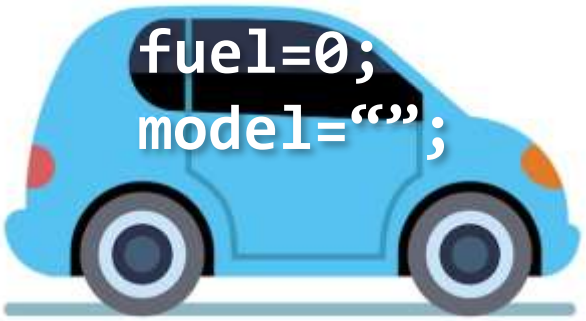
```
Car car1, car2, car3;

//assume class has a
setter
car1.setFuel(50);
car2.setFuel(30);
```



car1



car2

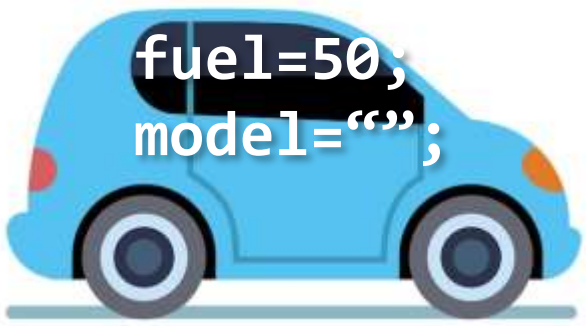


car3

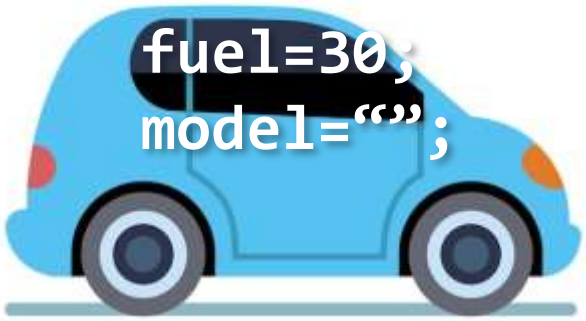
Class Instance Variables (Attributes)

```
class Car {
      int fuel;
      string model;
public:
Car() { //constructor
      fuel = 0;
      model = “”;}
};
```

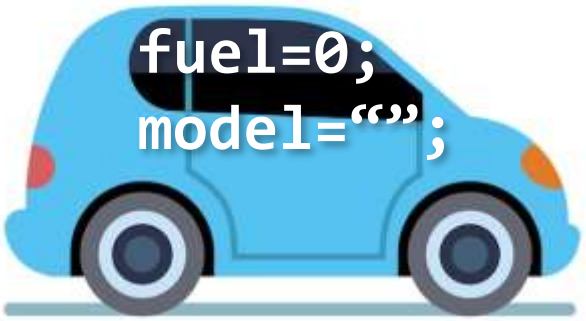
```
Car car1, car2, car3;

//assume class has a
setter
car1.setFuel(50);
car2.setFuel(30);
car3.setFuel(84);
```



car1



car2



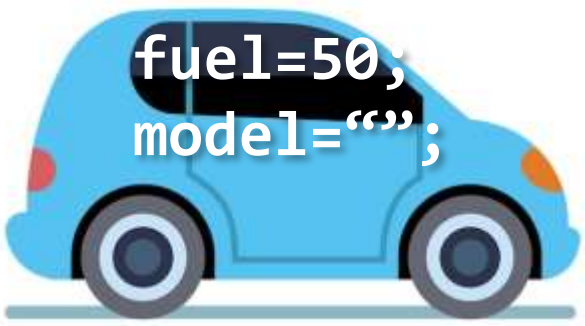
car3

Class Instance Variables

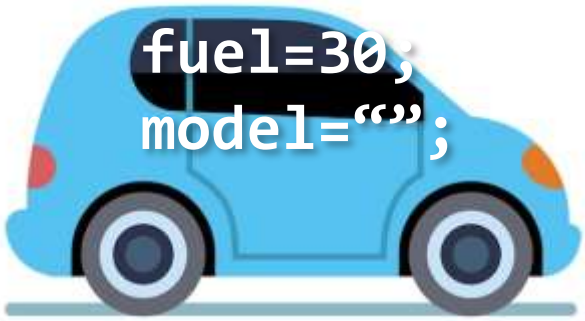
Each class object has **its own copy of the class's member variables.**

```
class Car {
      int fuel;
      string model;
public:
Car() { //constructor
      fuel = 0;
      model = “”;}
};
```

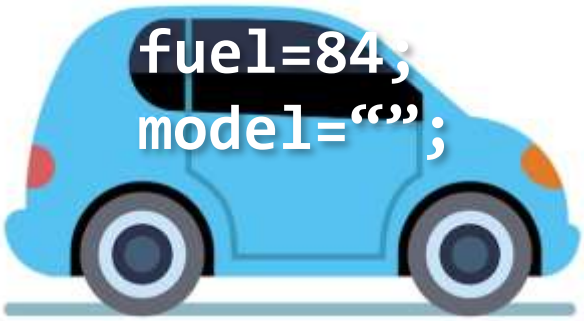
```
Car car1, car2, car3;

//assume class has a setter
car1.setFuel(50);
car2.setFuel(30);
car3.setFuel(84);
```



car1



car2



car3

`static` Class Members

- Class members include instance variables and functions
- It is possible to create a member variable or member function that does ***not belong to any instance of a class***.
- Such members are known as a static member variables and static member functions.
- You can think of static class members as belonging to the class instead of to an instance of the class

`static` Member Variables

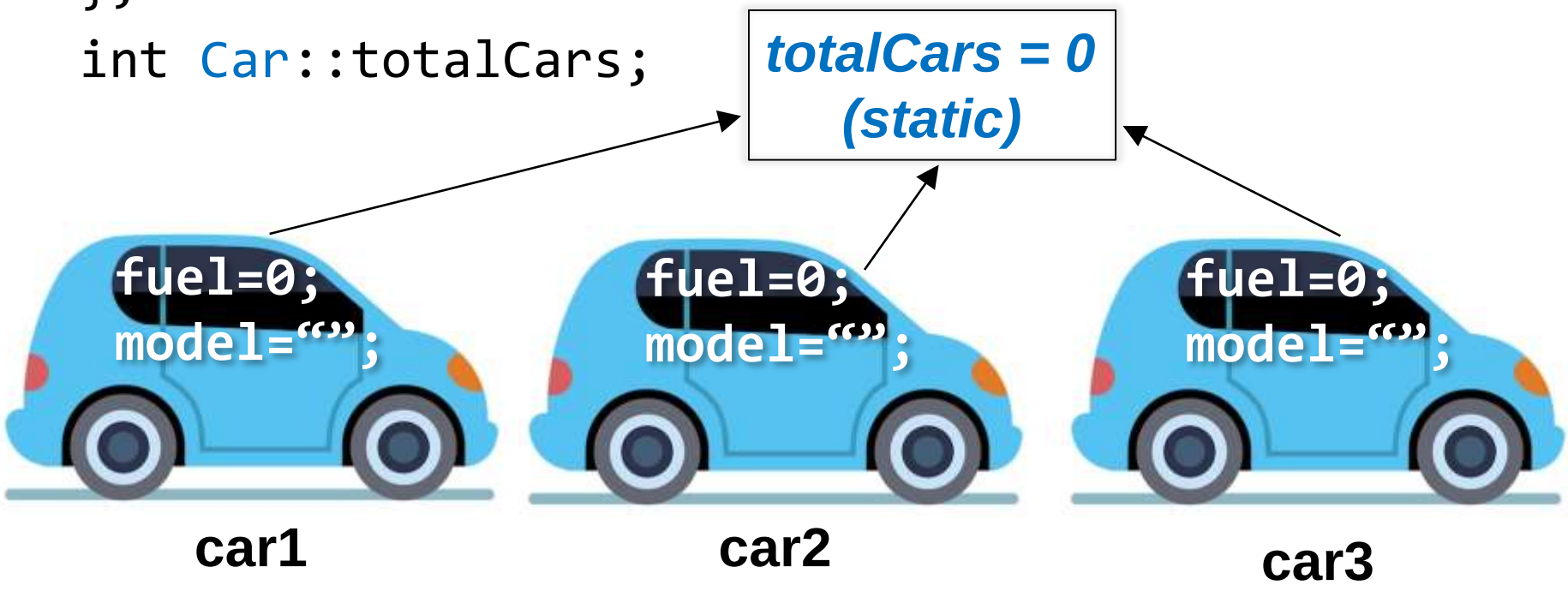
- When a member variable is ***static***, there will be only one copy of it in memory, regardless of the number of instances of the class that might exist.

- A single copy of a class's static member variable is shared by all objects of the class

- Static member variables exist even if no instances (objects) of the class exist

```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”; }
};
int Car::totalCars;
```

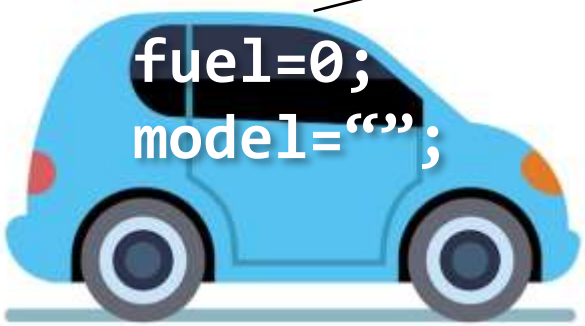
```
Car car1,car2,car3;
```



```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

```
Car car1,car2,car3;
```

totalCars = 0 (static)

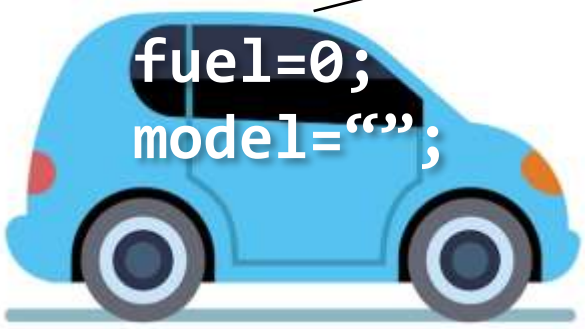


car1

```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

```
Car car1,car2,car3;
```

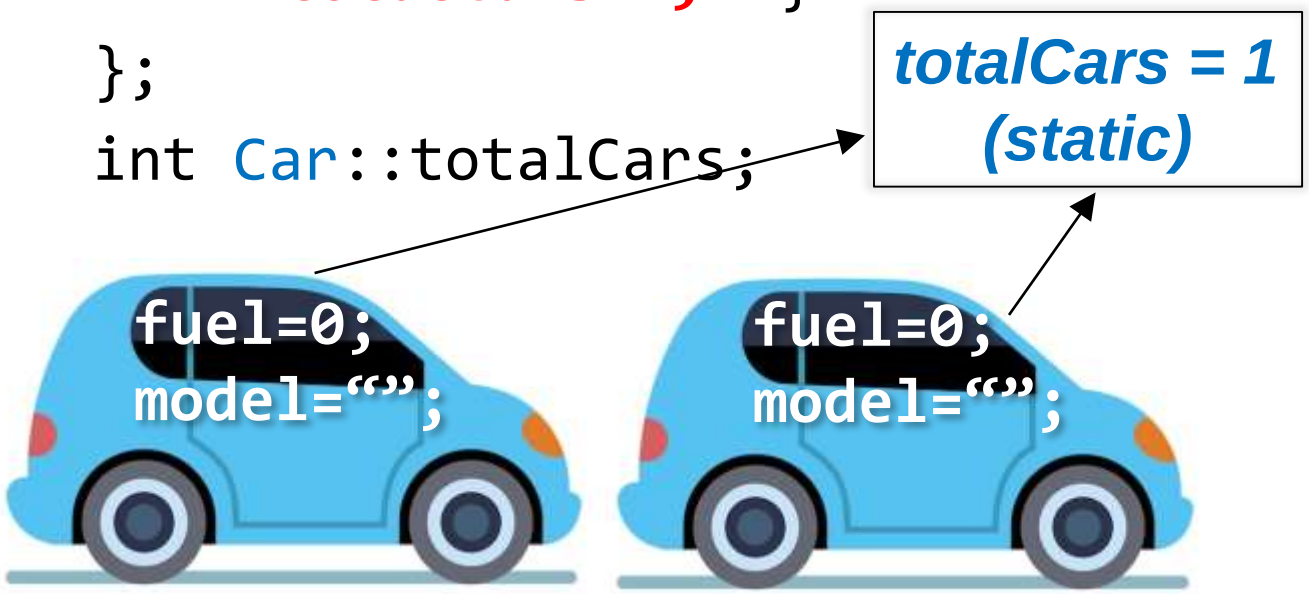
totalCars = 1 (static)



car1

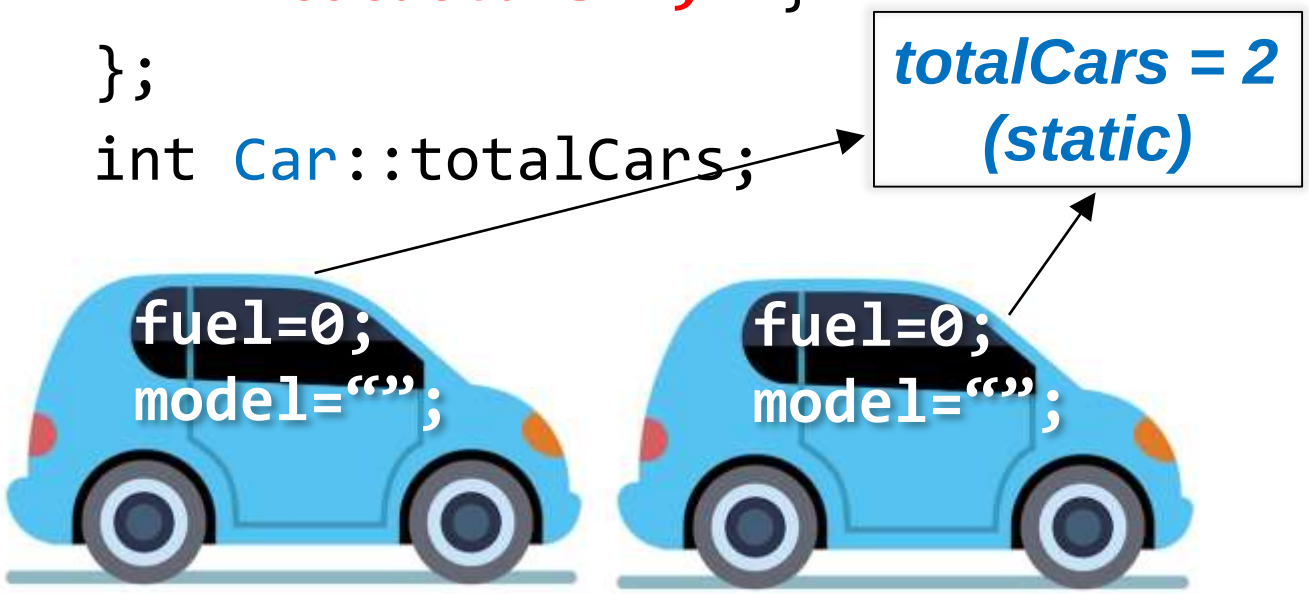
```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

```
Car car1,car2,car3;
```



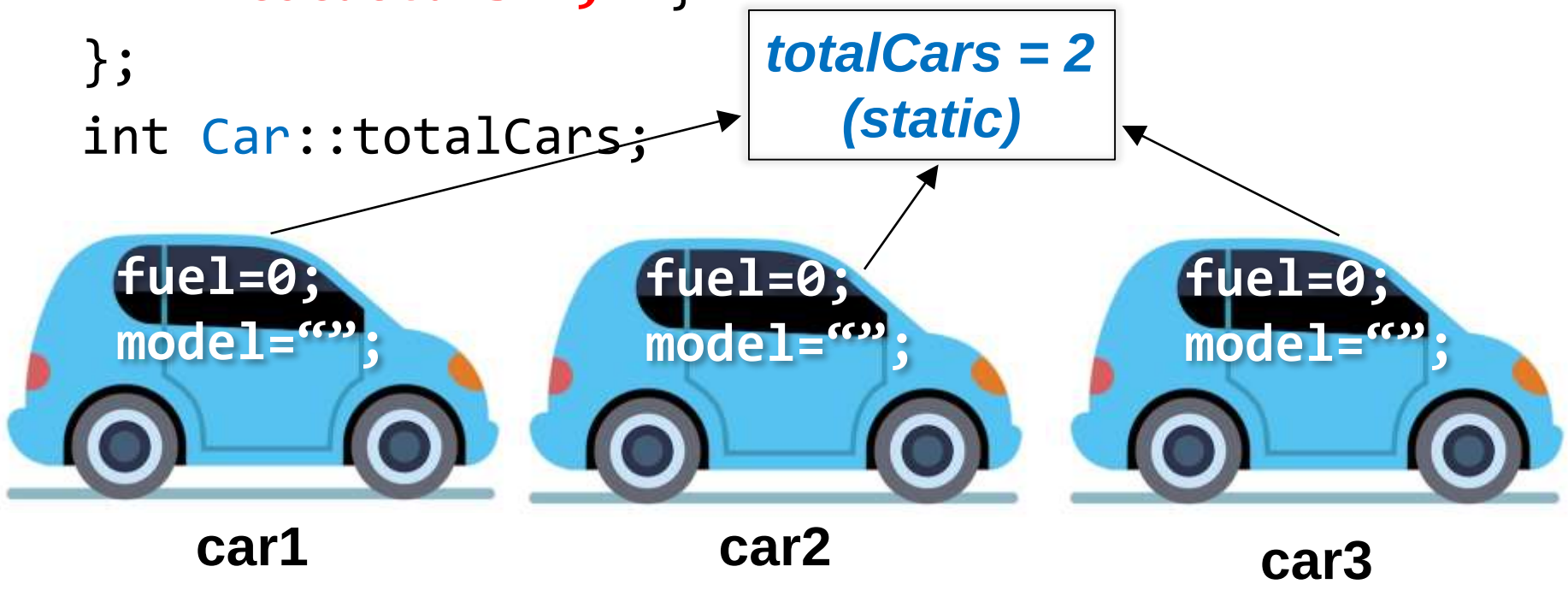
```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

```
Car car1,car2,car3;
```



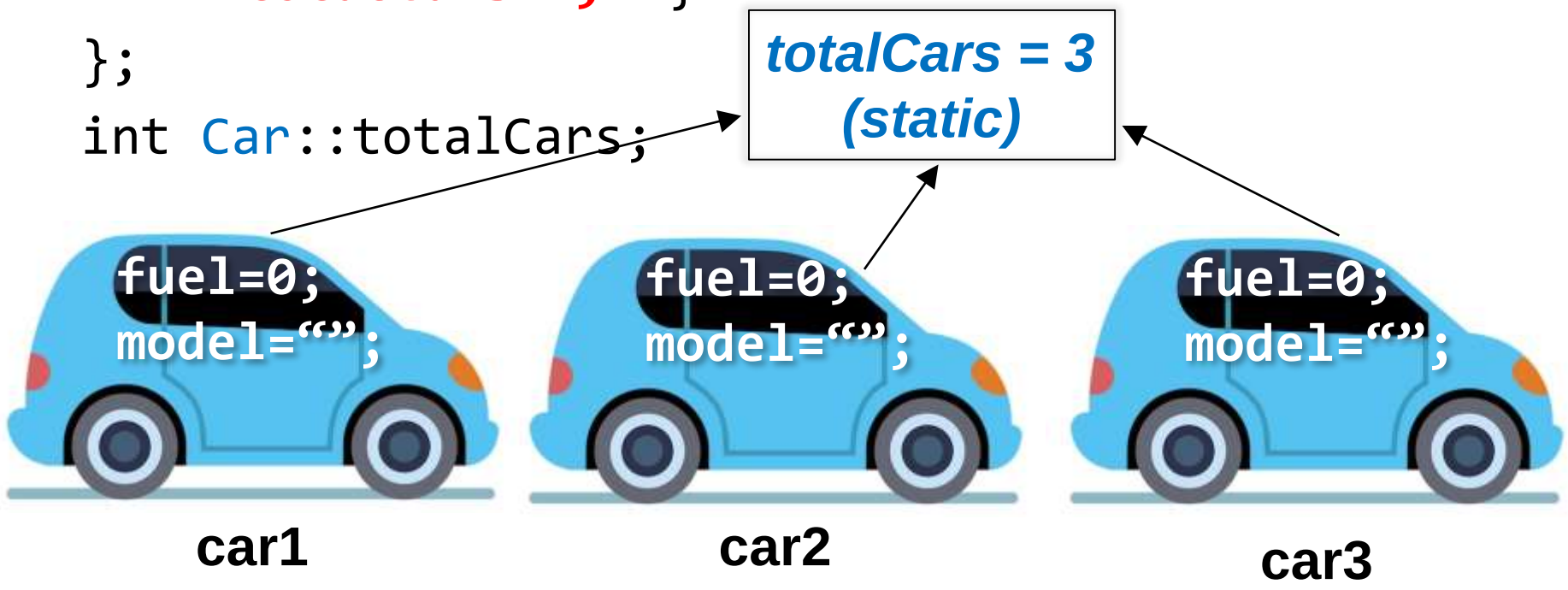
```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

```
Car car1,car2,car3;
```



```
class Car {
      int fuel;
      string model;
      static int totalCars;
public:
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
};
int Car::totalCars;
```

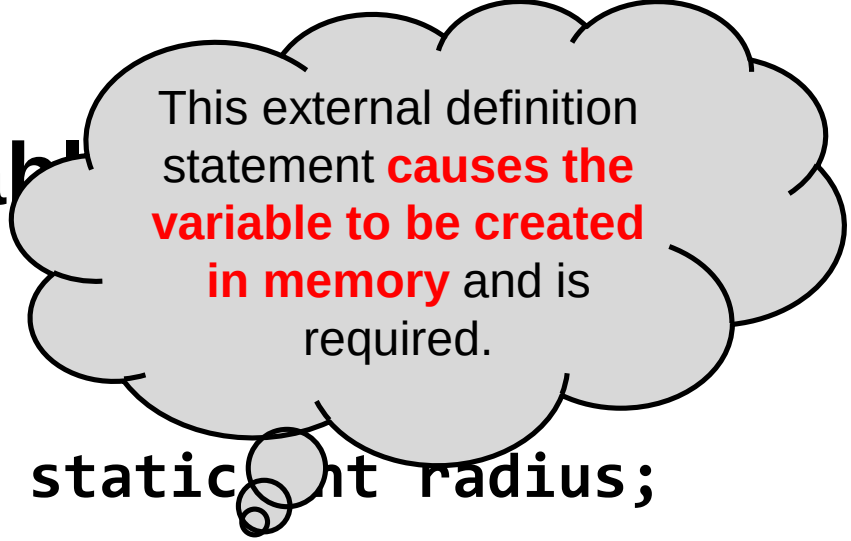
```
Car car1,car2,car3;
```



`static` Member Variables

- Shared by all objects of a class
- Efficient, when a single copy of data is enough
  - Only the static variable has to be updated
- May seems like global variables, but have class scope
  - only accessible to objects of same class
- Exist even if no instances (objects) of the class exist
- Can be public or private

`static` Member Variab



- **Two-Step Procedure:**
  1. Declare (Inside Class): `static int radius;`
  2. Define (Outside Class): `int Circle::radius;`

- Static Variables
  - Default Initialization: 0 or Null (for pointers)
  - Initialization: user defined value
  - Initialization is made just once, at compile time.
  - Accessibility: Private or Public

`Public static` Member Variables

- Can be accessed using class name:

```
cout << Car::totalCars;
```

- Can be accessed via any class' object:

```
cout << car1.totalCars;
```

- Can be accessed via Non-Static member functions:
- Must create object first

```
cout << car1.getCars();
```

- Can be accessed via Static member functions:

```
cout << Car::getTotalCars();
cout << car1.getTotalCars(); //public static
```

`Private static` Member Variables

- Cannot be accessed using class name:

```
// ERROR ⍰   cout << Car::totalCars;
```

- Cannot be accessed via class' object:

```
// ERROR ⍰  cout << car1.totalCars;
```

- Can be accessed via Non-Static member functions:

  Must create object first

```
cout << car1.getCars();
```

- Can be accessed via Static member functions:

```
cout << Car::getTotalCars();
cout << car1.getTotalCars(); //public static
```

`static` Member Variables

- The lifetime of a class's static member variable is the lifetime of the program

- This means that a class's static member variables come into existence before any instances of the class are created

`static` Member Functions

- Static member functions can operate only on static member variables.
- You can think of static member functions as belonging to the class instead of to an instance of the class.
- Static member functions can be called without creating any class objects

`static` Member Functions

```
class Car {
      int fuel;
      string model;
      static int totalCars;
Car() { //constructor
      fuel = 0;
      model = “”;
      totalCars++;  }
static int getTotalCars(){
      return totalCars; //only access static members
      }
};
int Car::totalCars;
```

`static` Member Functions

- **Non-static function**:
  - Can access: all class members

- **Static functions:**
  - Can access: static data and static functions
  - Cannot access: non-static data, non-static functions, and *this* pointer

`Public` `static` Member Functions

- Can be invoked using any class object:

```
cout << car1.getTotalCars();
```

- Can be invoked using class name:

```
cout << Car:: getTotalCars();
```

`Private static` Member Functions

- Cannot be invoked using class objects

```
//ERROR ⍰  cout << e1.getCount();
```

- Cannot be invoked using class name

```
//ERROR ⍰  cout << Employee::getCount();
```

- Can be invoked ***within class***:
  - Static member functions
  - Non-static member functions

```
1  // Fig. 7.9: employ1.h
2  // An employee class
3  #ifndef EMPLOY1_H
4  #define EMPLOY1_H
5
6  class Employee {
7  public:
8     Employee( const char*, const char* );  //
9     ~Employee();                           //
10    const char *getFirstName() const;  // return
11    const char *getLastName() const;   // return
12
13    // static member function
14    static int getCount();  // return # objects
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count;  // number of objects
22 };
23
24 #endif
```

static member function and variable declared.

```
25  // Fig. 7.9: employ1.cpp
26  // Member function definitions for class Employee
27  #include <iostream>
28
29  using std::cout;
30  using std::endl;
31
32  #include <cstring>
33  #include <cassert>
34  #include "employ1.h"
35
36  // Initialize the static data member
37  int Employee::count = 0;
38
39  // Define the static member function that
40  // returns the number of employee objects instantiated.
41  int Employee::getCount() { return count; }
42
43  // Constructor dynamically allocates space for the
44  // first and last name and uses strcpy to copy
45  // the first and last names into the object
46  Employee::Employee( const char *first, const char *last
47  {
48     firstName = new char[ strlen( first ) + 1 ];
49
50     strcpy( firstName, first );
51
52     lastName = new char[ strlen( last ) + 1 ];
53
54     strcpy( lastName, last );
55
56     ++count;  // increment static count of employees
```

`static` data member `count` and function `getCount( )` initialized (required).

`static` data member `count` changed when a constructor/destructor called.

```
57          cout << "Employee constructor for " << firstName
58               << ' ' << lastName << " called." << endl;
59       }
60
61       // Destructor deallocates dynamically allocated memory
62       Employee::~Employee()
63       {
64          cout << "~Employee() called for " <<
65               << ' ' << lastName << endl;
66          delete [] firstName;  // recapture memory
67          delete [] lastName;   // recapture memory
68          --count;   // decrement static count of employees
69       }
70
71       // Return first name of employee
72       const char *Employee::getFirstName() const
73       {
74          // Const before return type prevents client from
75          // private data. Client should copy returned string
76          // destructor deletes storage to prevent undefined
77          return firstName;
78       }
79
80       // Return last name of employee
81       const char *Employee::getLastName() const
82       {
83          // Const before return type prevents client from
84          // private data. Client should copy returned string
85          // destructor deletes storage to prevent undefined
86          return lastName;
87       }
```

static data member **count** changed when a **constructor/destructor called**.

Count decremented because of destructor calls from **delete**.

```
88  // Fig. 7.9: fig07_09.cpp
89  // Driver to test the employee class
90  #include <iostream>
91
92
93  using std::endl;
94
95  #include "employ1.h"
96
97  int main()
98  {
99     cout << "Number of employees before
100         << Employee::getCoun
101
102    Employee *e1Ptr = new Employee( "Susan", "Baker"
103    Employee *e2Ptr = new Employee( "Robert",
104
105    cout << "Number of employees after instantiation
106         << e1Ptr->getCount();
107
108    cout << "\n\nEmployee 1: "
109         << e1Ptr->getFirstName()
110         << " " << e1Ptr->get
111         << "\nEmployee 2: "
112         << e2Ptr->getFirstName()
113         << " " << e2Ptr->getLastName() << "\n\n";
114
115    delete e1Ptr;    // free me
116    e1Ptr = 0;
117    delete e2Ptr;    // free memory
118    e2Ptr = 0;
```

count incremented because of constructor calls from new.

If no Employee objects exist getCount must be accessed using the class name and (::).

Number of employees before instantiation is 0

e2Ptr->getCount() or Employee::getCount() would also work.

Number of employees after instantiation is 2

Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones

```
119
120           cout << "Number of employees after deletion is
"
121                << Employee::getCount() << endl;
122
123           return 0;
124       }
```

count back to zero.

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

`static` Class Members

Calling a static member function of a class using the object

Calling a static member function of a class using the class name

this Pointer

- The *this* pointer is a special built-in pointer that is available to a class's **non-static** member functions.

- It always points to the instance of the class making the function call.

- The *this* pointer is passed as a hidden argument to all non-static member functions.

```
void setFuel(int fuel, Car* const this) {

    this->fuel = fuel;
}
```

this Pointer

```
 void setFuel(int fuel, Car* const this) {

      this->fuel = fuel;
}

Car car1, car2;
```

car1.setFuel(20); // this pointer points at car1
car2.setFuel(40); // this pointer points at car2

- *this* pointer is a constant pointer that references to the calling object.

Using the this Pointer

- Not part of the ***object itself***
- Can be used to access instance variables within member functions (including constructors and destructors)
- this pointer stores the address of the calling object
- For a member function print data member x, either

```
this->x;
```

or

```
(*this).x
```

Using the this Pointer

- Useful when function parameters and class member variables have the same name

```
//Parametrized Constructor
Car(int fuel, string model) {
      this->fuel = fuel;
      this->model = model;

}
```

Using the this Pointer

- Function returns a reference pointer to the same object

```
{ return *this; }
```

  - Other functions can operate on that pointer

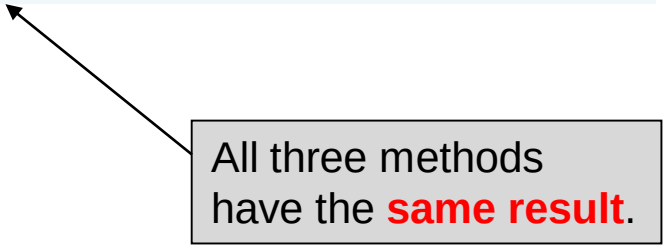
```
1  // Fig. 7.7: fig07_07.cpp
2  // Using the this pointer to refer to object
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  class Test {
9  public:
10    Test( int = 0 );              // default
11    void print() const;
12 private:
13    int x;
14 };
15
16 Test::Test( int a ) { x = a; } //default
17
18 void Test::print() const    // ( ) around
19 {
20    cout << "   using variable = " << x
21         << "\n using this pointer = " << this->x
22         << "\n using this with * = " << ( *this
23 }
24
25 int main()
26 {
27    Test testObject( 12 );
28
29    testObject.print();
30
31    return 0;
32 }
```

Printing **x** directly.

Print **x** using the arrow **->** operator off the **this** pointer.

Printing **x** using the dot (**.**) operator. Parenthesis required because dot operator has higher precedence than *****. Without, interpreted incorrectly as ***(this.x)**.

```
using variable                = 12
using this pointer  = 12
using this with *   = 12
```



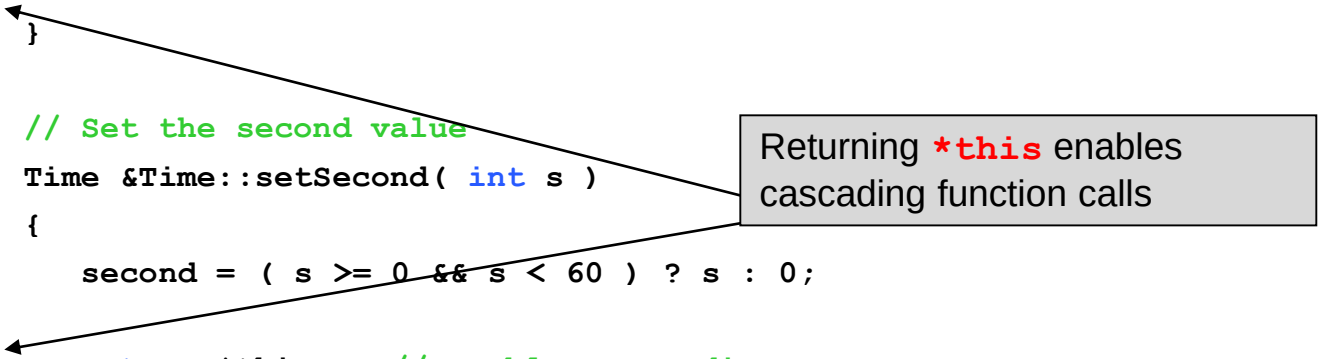
```
1  // Fig. 7.8: time6.h
2  // Cascading member function calls.
3
4  // Declaration of class Time.
5  // Member functions defined in time6.cpp
6  #ifndef TIME6_H
7  #define TIME6_H
8
9  class Time {
10 public:
11    Time( int = 0, int = 0, int = 0 );  // default
12
13    // set functions
14    Time &setTime( int, int, int ); // set hour,
15    Time &setHour( int );     // set hour
16    Time &setMinute( int );   // set minute
17    Time &setSecond( int );   // set secon
18
19    // get functions (normally declared c
20    int getHour() const;      // return ho
21    int getMinute() const;    // return mi
22    int getSecond() const;    // return second
23
24    // print functions (normally declared const)
25    void printMilitary() const;  // print military
26    void printStandard() const;  // print standard
27 private:
28    int hour;              // 0 - 23
29    int minute;            // 0 - 59
30    int second;            // 0 - 59
31 };
32
```

Notice the **`Time &`** - function returns a **reference to a `Time` object**.
Specify object in function definition.

```
34  // Fig. 7.8: time.cpp
35  // Member function definitions for Time class.
36  #include <iostream>
37
38  using std::cout;
39
40  #include "time6.h"
41
42  // Constructor function to initialize private data.
43  // Calls member function setTime to set variables.
44  // Default values are 0 (see class definition).
45  Time::Time( int hr, int min, int sec )
46     { setTime( hr, min, sec ); }
47
48  // Set the values of hour, minute, and second.
49  Time &Time::setTime( int h, int m,
50  {
51     setHour( h );
52     setMinute( m );
53     setSecond( s );
54     return *this;   // enables cascading
55  }
56
57  // Set the hour value
58  Time &Time::setHour( int h )
59  {
60     hour = ( h >= 0 && h < 24 ) ? h : 0;
61
62     return *this;   // enables cascading
63  }
64
```

Returning *this enables cascading function calls

```
65          // Set the minute value
66          Time &Time::setMinute( int m )
67          {
68             minute = ( m >= 0 && m < 60 ) ? m : 0;
69
70             return *this;    // enables cascading
71          }
72
73          // Set the second value
74          Time &Time::setSecond( int s )
75          {
76             second = ( s >= 0 && s < 60 ) ? s : 0;
77
78             return *this;    // enables cascading
79          }
80
81          // Get the hour value
82          int Time::getHour() const { return hour; }
83
84          // Get the minute value
85          int Time::getMinute() const { return minute; }
86
87          // Get the second value
88          int Time::getSecond() const { return second; }
89
90          // Display military format time: HH:MM
91          void Time::printMilitary() const
92          {
93             cout << ( hour < 10 ? "0" : "" ) << hour << ":"
94                  << ( minute < 10 ? "0" : "" ) << minute;
```



Using the this Pointer

- **Example of cascaded member function calls:**
  - Member functions **setHour**, **setMinute**, and **setSecond** all return *this (reference to an object)
  - For object **t**, consider:

    ```
    t.setHour(1).setMinute(2).setSecond(3);
    ```

  - Executes `t.setHour(1),` returns *this (reference to object) and the expression becomes

    ```
    t.setMinute(2).setSecond(3);
    ```

  - Executes `t.setMinute(2),` returns reference and becomes

    ```
    t.setSecond(3);
    ```

  - Executes `t.setSecond(3)`, returns reference and becomes

    ```
    t; (Has no effect)
    ```

```
95      }
96
97      // Display standard format time: HH:MM:SS AM
98      void Time::printStandard() const
99      {
100        cout << ( ( hour == 0 || hour == 12 ) ? 12 :
101             << ":" << ( minute < 10 ? "0" : "" ) <<
102             << ":" << ( second < 10 ? "0" : "" ) <<
103             << ( hour < 12 ? " AM" : " PM" );

105     // Fig. 7.8: fig07_08.cpp
106     // Cascading member function calls together
107     // with the this pointer
108     #include <iostream>
109
110     using std::cout;
111     using std::endl;
112
113     #include "time6.h"
114
115     int main()
116     {
117        Time t;
118
119        t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
120        cout << "Military time
121        t.printMilitary();
122        cout << "\nStandard ti
123        t.printStandard();
124
125        cout << "\n\nNew standard time: ";
126        t.setTime( 20, 20, 20 ).printStandard();
```

printStandard does not return a reference to an object.

Notice cascading function calls.

Cascading function calls. **printStandard** must be called after **setTime** because **printStandard** does not return a reference to an object.
t.printStandard().setTime(); would cause an error.

```
127         cout << endl;
128
129         return 0;
130      }
```

```
Military time: 18:30
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Member Initializer List

- You have used initializer lists with arrays

```
int arr[] = {1,2,3,4,5};
```

- The data members of a class can also be initialized using an initializer list with the constructor

Member Initializer List

```
class Test {
        int marks;
        int total;

public:
//inline with initializer list
Test() : marks(0),total(0) {} //default constructor

Test(int m, int t) : marks(m), total(t) {}

int getMarks() {
        return marks;
    }
int getTotal() {
        return total;
    }
};
```

Member Initializer List

```
class Test {
        int marks;
        int total;

public:
//inline with initializer list
Test() : marks(0),total(0) {} //default constructor

Test(int m, int t) : marks(m), total(t) {}

int getMarks() {
        return marks;
    }
int getTotal() {
        return total;
    }
};
```

Member Initializer List

```
class Test {
        int marks;
        int total;

public:
        Test(); //prototype
        Test(int, int ); //prototype
int getMarks() {
        return marks;
}
int getTotal() {
        return total;
        }
};
//out of line with initializer list
Test::Test() :marks(0), total(0) {}
Test::Test(int m,int t): marks(m), total(t) {}
```

Member Initializer List (Non-const Members)

```
class Point {
private:
    int x;
    int y;
public:
    Point(int i = 2, int j = 3):y(i) {x=j;}
    /* The above use of Initializer list is optional as the  constructor can also
    be written as:
    Point(int i = 0, int j = 0) {
            x = i;
            y = j;
    } */

    int getX() const {return x;}
    int getY() const {return y;}
};

int main() {
  Point t1(10, 15);
  cout<<"x = "<<t1.getX()<<", ";
  cout<<"y = "<<t1.getY();
  return 0;
}
```

When to use Member Initializer List

- You can use initializer list to initialize member variables but it is not necessary

- But you **must use** an initializer list for
  - Initializing a constant data member
  - Initializing a reference member

`const` Class Members

- As with member functions, data members can also be ***const***

- Constant members **must be initialized using member initializer list**

Member Initializer List (non-static const)

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

class Test {
    const int t;
public:
    Test(int x):t(x) {} //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}
```

Member Initializer List (References)

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

class Test {
    int &cRef;
public:
    Test(int &ref):cRef(ref) {}  //Initializer list must be used
    int getRef() { return cRef; }
};

int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getRef()<<endl;
    x = 30;
    cout<<t1.getRef()<<endl;
    return 0;
}
```

Member Initializer List (parameter name same as data member)

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

class A {
    int i;
public:
    A(int );
    int getI() const { return i; }
};

A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
    this->i = i;
}
*/

int main() {
    A a(10);
    cout<<a.getI();
    return 0;
}
```

`const` Objects

- Read-only objects
  - Object data members can only be read, NO write/update of data member allowed
  - Requires all member functions be const (except constructors and destructors)
  - const object must be initialized (using constructors) at the time of object creation

```
const Account inv("YMCA, FL", 5555, 5000.0);
```

const Objects

- const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes
  - So the constructor and destructor can modify the object

friend Functions and *friend* Classes

- A *friend* is a function or class that is not a member of a class, but has access to the private members of the class
- In other words, a friend function is treated as if it were a member of the class
- A friend function can be a regular stand-alone function, or it can be a member of another class.
- An entire class can also be declared a friend of another class

friend Functions and *friend* Classes

- To declare a **friend** function
  - Type **friend** before the function prototype in the class that is giving friendship

```
friend int myFunction( int x );
```

  - should appear in the class giving friendship

- To declare a **friend** class
  - Type **friend class Classname** in the class that is giving friendship
  - if **ClassOne** is granting friendship to **ClassTwo**,

```
friend class ClassTwo;
```

  - should appear in **ClassOne**'s definition

friend Functions and *friend* Classes

- **`friend`** functions and **`friend`** classes
  - Can access **`private`** members of another class

- Properties of friendship
  - Friendship is granted, not taken
  - Not symmetric (if **`B`** a **`friend`** of **`A`**, **`A`** not necessarily a **`friend`** of **`B`**)
  - Not transitive (if **`A`** a **`friend`** of **`B`**, **`B`** a **`friend`** of **`C`**, **`A`** not necessarily a **`friend`** of **`C`**)

```
1  // Fig. 7.5: fig07_05.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // Modified Count class
9  class Count {
10    friend void setA( Count &, int ); // friend
11 public:
12    Count() { a = 0; }                  // constructor
13    void print() const { cout << x << endl; }  //
14 private:
15    int a;  // data member
16 };
17
18 // Can modify private data
19 // setX is declared as a friend function of Count
20 void setA( Count
21 {
22    c.x = val;  // legal: setX is a friend of Count
23 }
24
25 int main()
26 {
27    Count counter;
28
29    cout << "counter.a after instantiation: ";
30    counter.print();
```

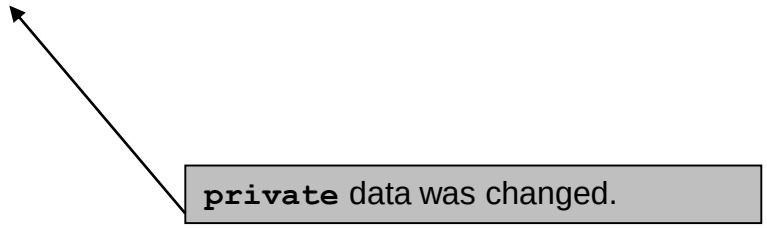
setA a **friend** of class **Count** (can access **private** data).

setA is defined normally and is not a member function of **Count**.

Changing **private** variables allowed.

```
31            cout << "counter.a after call to setA friend
function: ";
32            setA( counter, 8 );  // set a with a friend
33            counter.print();
34            return 0;
35         }
```

```
counter.a after instantiation: 0
counter.a after call to setA friend function: 8
```

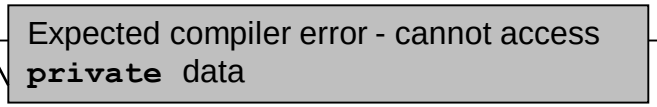


```
1  // Fig. 7.6: fig07_06.cpp
2  // Non-friend/non-member functions cannot access
3  // private data of a class.
4  #include <iostream>
5
6  using std::cout;
7  using std::end
8
9  // Modified Count class
10 class Count {
11 public:
12    Count() { x = 0; }                    //
13    void print() const { cout << x << endl; }  //
14 private:
15    int x;  // data member
16 };
17
18 // Function tries to modify private data of Count,
19 // but cannot because it is not a friend of Count.
20 void cannotSetA( Count &c, int val )
21 {
22    c.x = val;  // ERROR: 'Count::x' is not
23 }
24
25 int main()
26 {
27    Count counter;
28
29    cannotSetX( counter, 3 ); // cannotSetX is not a
30    return 0;
31 }
```

cannotSetA is not a **friend** of class **Count**. It cannot access **private** data.

cannotSetA tries to modify a **private** variable…

```
Compiling...
Fig07_06.cpp
D:\books\2000\cpphtp3\examples\Ch07\Fig07_06\Fig07_06.cpp(22) :
   error C2248: 'x' : cannot access private member declared in
   class 'Count'
        D:\books\2000\cpphtp3\examples\Ch07\Fig07_06\
        Fig07_06.cpp(15) : see declaration of 'x'
Error executing cl.exe.

test.exe - 1 error(s), 0 warning(s)
```



friend Functions and *friend* Classes

- Friends should be used only for limited purpose
- Too many functions declared as friends with private data access, lessens the value of encapsulation