Object-Oriented Programming

- Class Definition
- Class Examples
- Objects
- Constructors
- Destructors

Class

- **The class is the cornerstone of C++**
  - It makes possible encapsulation, data hiding and inheritance
- **Type**
  - Concrete representation of a concept
    - Eg. **float** with operations like -, *, + (math real numbers)
- **Class**
  - A user defined type
  - Consists of both data and methods
  - Defines properties and behavior of that type
- **Advantages**
  - Types matching program concepts
    - Game Program (Explosion type)
  - Concise program
  - Code analysis easy
  - Compiler can detect illegal uses of types
- **Data Abstraction**
  - Separate the implementation details from its essential properties

Class & Object

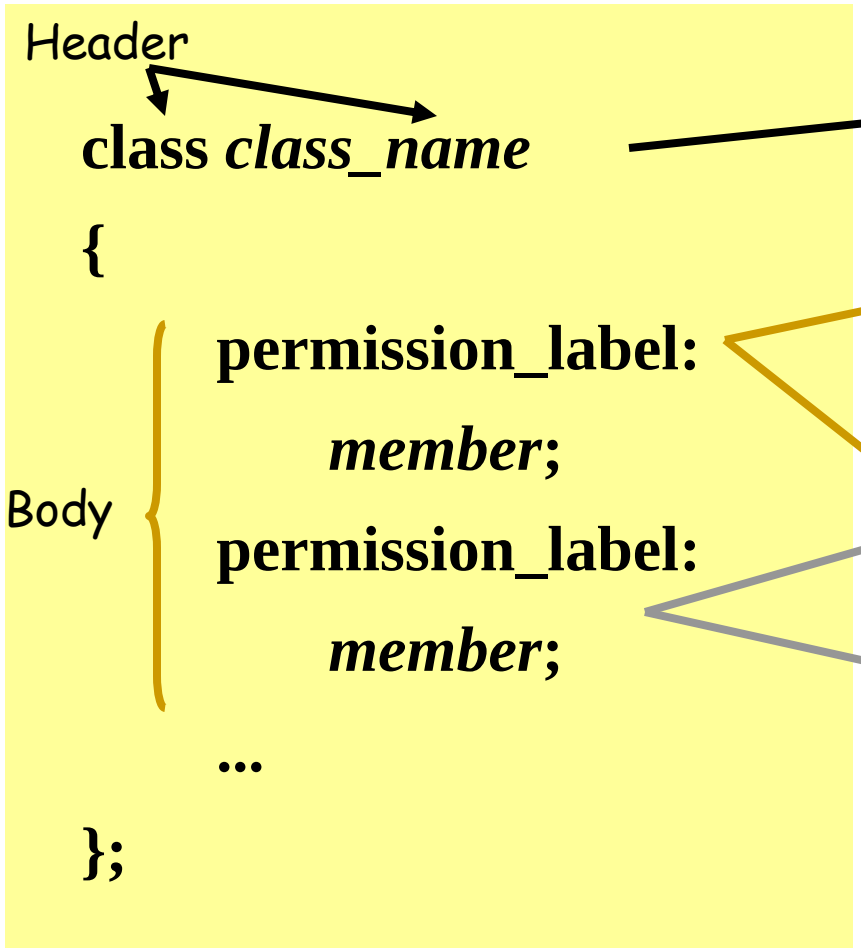
```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

Objects: Instance of a class

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
```

```
int   a;
```

Define a Class Type




```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

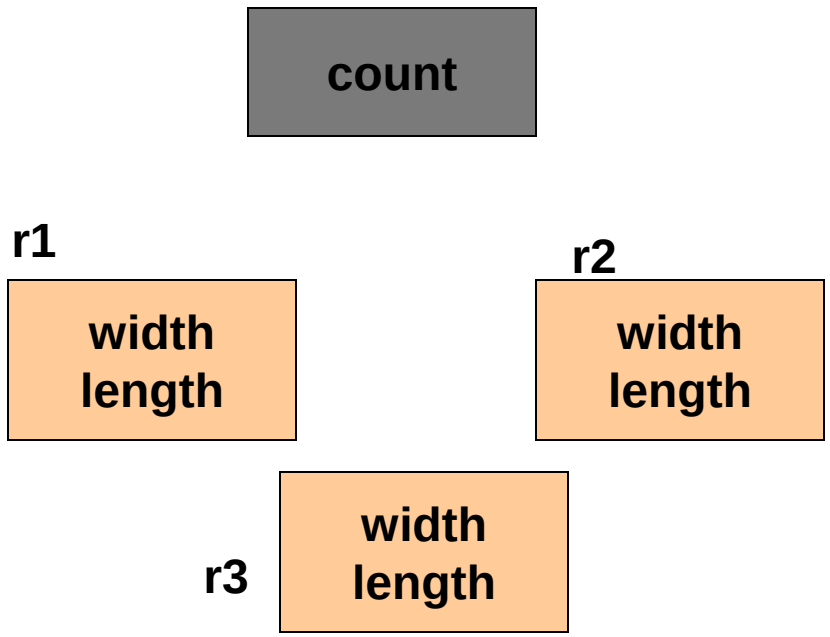
Class Definition – Data Members

- **Can be of any type, built-in or user-defined**
- ***Non-static* data member**
  - Each class object has its own copy
- ***Static* data member**
  - Acts as a global variable
  - One copy per class type, e.g. counter

Static Data Member

```
class Rectangle
{
   private:
      int width;
      int length;
      static int count;
   public:
      void set(int w, int l);
      int area();
   }
```

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
```



Class Definition – Member Functions

- **Used to**
  - access the values of the data members (**accessor**)
  - perform operations on the data members (**implementor**)
- **Are declared inside the class body**
- **Their definition can be placed inside the class body, or outside the class body**
- **Can access both public and private members of the class**
- **Can be referred to using dot or arrow member access operator**

Define a Member Function

```
class Rectangle
{
    private:
      int width, length;
    public:
      void set (int w, int l);
      int area() {return width*length; }
};
```

class name

member function name

inline

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

scope operator

```
r1.set(5,8);

rp->set(8,10);
```

Class Definition – Member Functions

- **const member function**
  - declaration
    - *return_type func_name* (*para_list*) const;
  - definition
    - *return_type func_name* (*para_list*) const { … }
    - *return_type class_name* :: *func_name* (*para_list*) const { … }
  - Makes no modification about the data members (safe function)
  - It is illegal for a const member function to modify a class data member

Const Member Function

```
class  Time
{
  private :
     int     hrs, mins, secs ;

  public :
     void     Write ( )  const ;
} ;
```

function declaration

function definition

```
void Time :: Write( ) const
{
    cout <<hrs << ":" << mins << ":" << secs << endl;
}
```

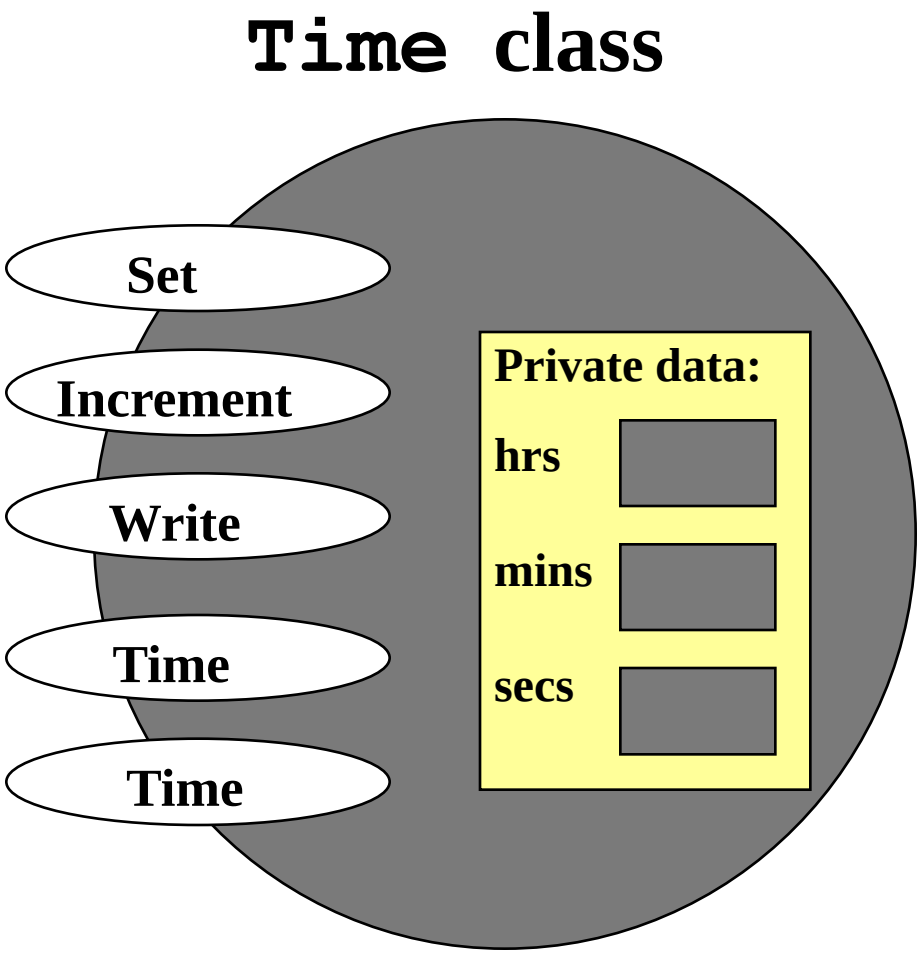
Class Definition – Access Control

- **Information hiding**
  - To prevent the internal representation from direct access from outside the class
- **Access Specifiers**
  - public
    - may be accessible from anywhere within a program
  - private
    - may be accessed only by the member functions, and friends of this class
  - protected
    - acts as public for derived classes
    - behaves as private for the rest of the program

Class 'Time' Specification

```
class  Time
{

  public :

    void     Set ( int  hours , int  minutes , int  seconds ) ;
    void       Increment ( ) ;
    void       Write ( )  const ;
    Time    ( int  initHrs, int  initMins,  int  initSecs ) ;   // constructor
    Time    ( ) ;                                                // default constructor

  private :

    int            hrs ;
    int            mins ;
    int             secs ;
} ;
```

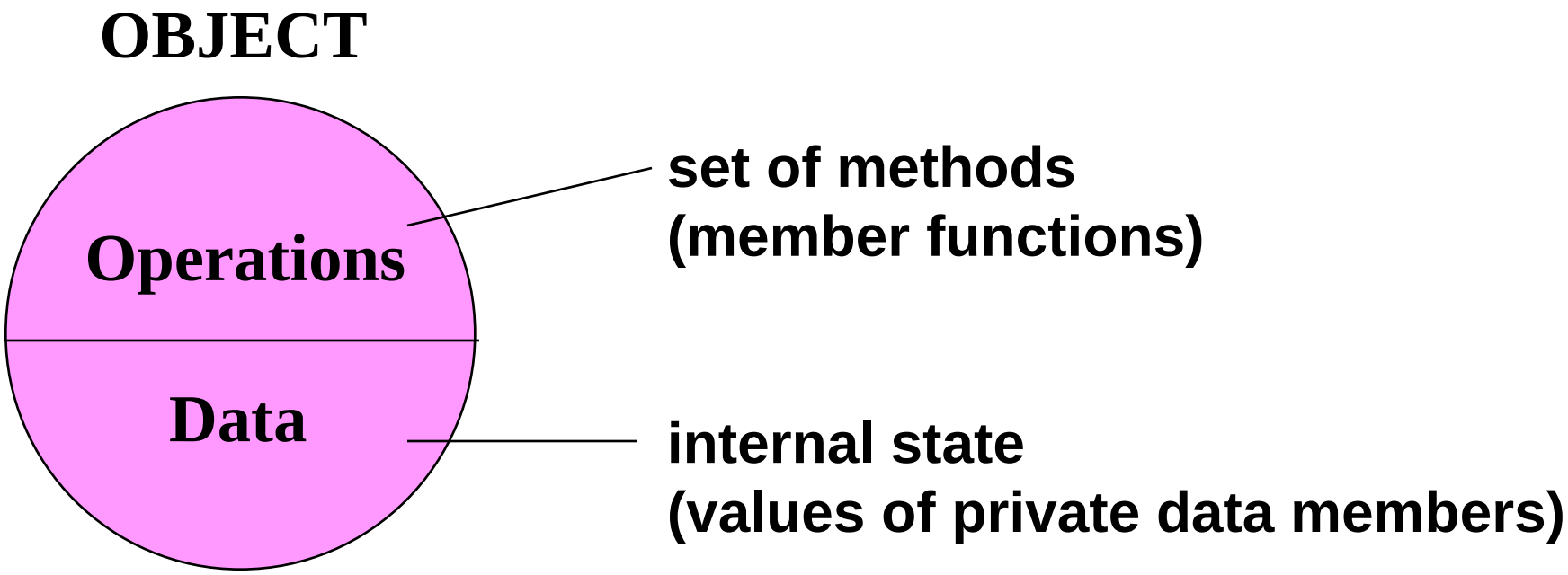
Class Interface Diagram



Class Definition – Access Control

- **The default access specifier is *private***
- **The data members are usually private or protected**
- **A private member function is a helper, may only be accessed by another member function of the same class (exception *friend* function)**
- **The public member functions are part of the *class interface***
- **Each access control section is optional, repeatable, and sections may occur in any order**

Object


OBJECT
Operations
Data
set of methods
(member functions)
internal state
(values of private data members)

Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

```
#include <iostream.h>

class circle
{
   private:
      double radius;

   public:
      void store(double);
      double area(void);
      void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
   radius = r;
}
double circle::area(void)
{
   return 3.14*radius*radius;
}
void circle::display(void)
{
   cout << “r = “ << radius << endl;
}
```

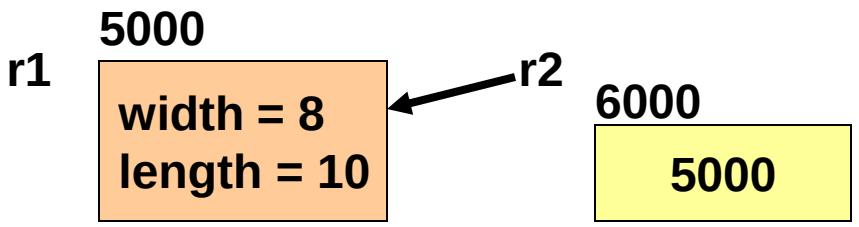
```
int main(void) {
    circle c;   // an object of circle class
    c.store(5.0);
    cout << "The area of circle c is " << c.area() << endl;
    c.display();
}
```

Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);        //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);       //arrow notation
}
```

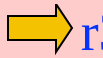


Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r3 is dynamically allocated

```
main()
{
    Rectangle *r3;
    r3 = new Rectangle();

    r3->set(80,100);     //arrow notation

    delete r3;
    r3 = NULL;
}
```



r3

6000

NULL

Object Initialization

1. By Assignment

```
#include <iostream.h>

class circle
{
  public:
    double radius;
};
```

- Only work for public data members
- No control over the operations on data members

```
int main()
{
   circle c1;                  // Declare an instance of the class circle
   c1.radius = 5;              // Initialize by assignment

}
```

Object Initialization

```
#include <iostream.h>

class circle
{
  private:
     double radius;

   public:
     void set (double r)
               {radius = r;}
     double get_r ()
               {return radius;}

};
```

2. By Public Member Functions

```
int main(void) {
    circle c;              // an object of circle class
    c.set(5.0);            // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
       // access a private data member with an accessor
}
```

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(const Rectangle &r);
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

3. By Constructor

- Default constructor
- Copy constructor
- Constructor with parameters

They are publicly accessible

Have the same name as the class

There is no return type

Are used to initialize class data members

They have different signatures

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       void set(int w, int l);
       int area();
};
```

When a class is declared with no constructors, the compiler automatically assumes default constructor and copy constructor for it.

- Default constructor

```
Rectangle :: Rectangle() { };
```

- Copy constructor

```
Rectangle :: Rectangle (const
    Rectangle & r)
{
    width = r.width;  length = r.length;
};
```

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       void set(int w, int l);
       int area();
};
```

- Initialize with default constructor

```
Rectangle r1;

Rectangle *r3 = new Rectangle();
```

- Initialize with copy constructor

```
Rectangle r4;
r4.set(60,80);

Rectangle r5 = r4;
Rectangle r6(r4);

Rectangle *r7 = new Rectangle(r4);
```

Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle(int w, int l)
          {width =w; length=l;}
      void set(int w, int l);
      int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;     // error
```

- Initialize with constructor

```
Rectangle r5(60,80);
Rectangle *r6 = new Rectangle(60,80);
```

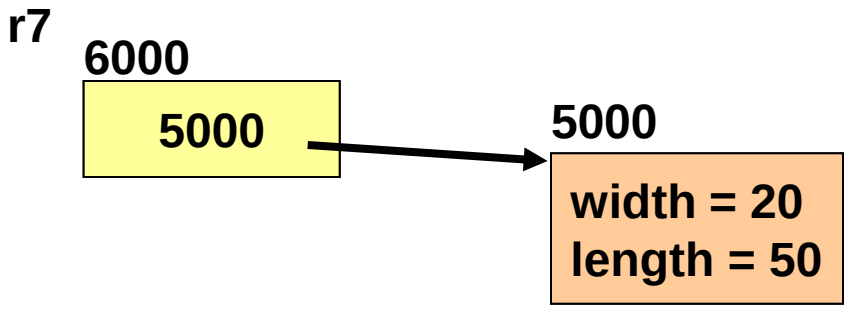
Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

Write your own constructors

```
Rectangle :: Rectangle()
{
    width = 20;
    length = 50;
};
```

```
Rectangle  *r7 = new Rectangle();
```



Object Initialization

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;
    public:
      Account();
      Account(const Account &a);
      Account(const char *person);
}
```

```
Account :: Account()
{
    name = NULL; balance = 0.0;
    id = 0;
};
```

With constructors, we have more control over the data members

```
Account :: Account(const Account &a)
{
    name = new char[strlen(a.name)+1];
    strcpy (name, a.name);
    balance = a.balance;
    id = a.id;
};
```

```
Account :: Account(const char *person)
{
    name = new char[strlen(person)+1];
    strcpy (name, person);
    balance = 0.0;
    id = 0;
};
```

Summary

- **An object can be initialized by a class constructor**
  - default constructor
  - copy constructor
  - constructor with parameters
- **Resources are allocated when an object is initialized**
- **Resources should be revoked when an object is about to end its lifetime**

Cleanup of an Object

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;  //unique
    public:
      Account();
      Account(const Account &a);
      Account(const char *person);
      ~Account();
}
```

Destructor

```
Account :: ~Account()
{
    delete[] name;
}
```

- Its name is the class name preceded by a ~ (tilde)
- It has no argument
- It is used to release dynamically allocated memory and to perform other "cleanup" activities
- It is executed automatically when the object goes out of scope

Putting them Together

```
class Str
{
    char *pData;
    int nLength;
  public:
     //constructors
    Str();
    Str(char *s);
    Str(const Str &str);

     //accessors
    char* get_Data();
    int get_Len();

     //destructor
     ~Str();
};
```

```
Str :: Str() {
    pData = new char[1];
    *pData = '\0';
    nLength = 0;
};
```

```
Str :: Str(char *s) {
    pData = new char[strlen(s)+1];
    strcpy(pData, s);
    nLength = strlen(s);
};
```

```
Str :: Str(const Str &str) {
    int n = str.nLength;
    pData = new char[n+1];
    nLength = n;
    strcpy(pData,str.pData);
};
```

Putting them Together

```
class Str
{
    char *pData;
    int nLength;
  public:
     //constructors
    Str();
    Str(char *s);
    Str(const Str &str);

     //accessors
    char* get_Data();
    int get_Len();

     //destructor
     ~Str();
};
```

```
char* Str :: get_Data()
{
    return pData;
};
```

```
int Str :: get_Len()
{
    return nLength;
};
```

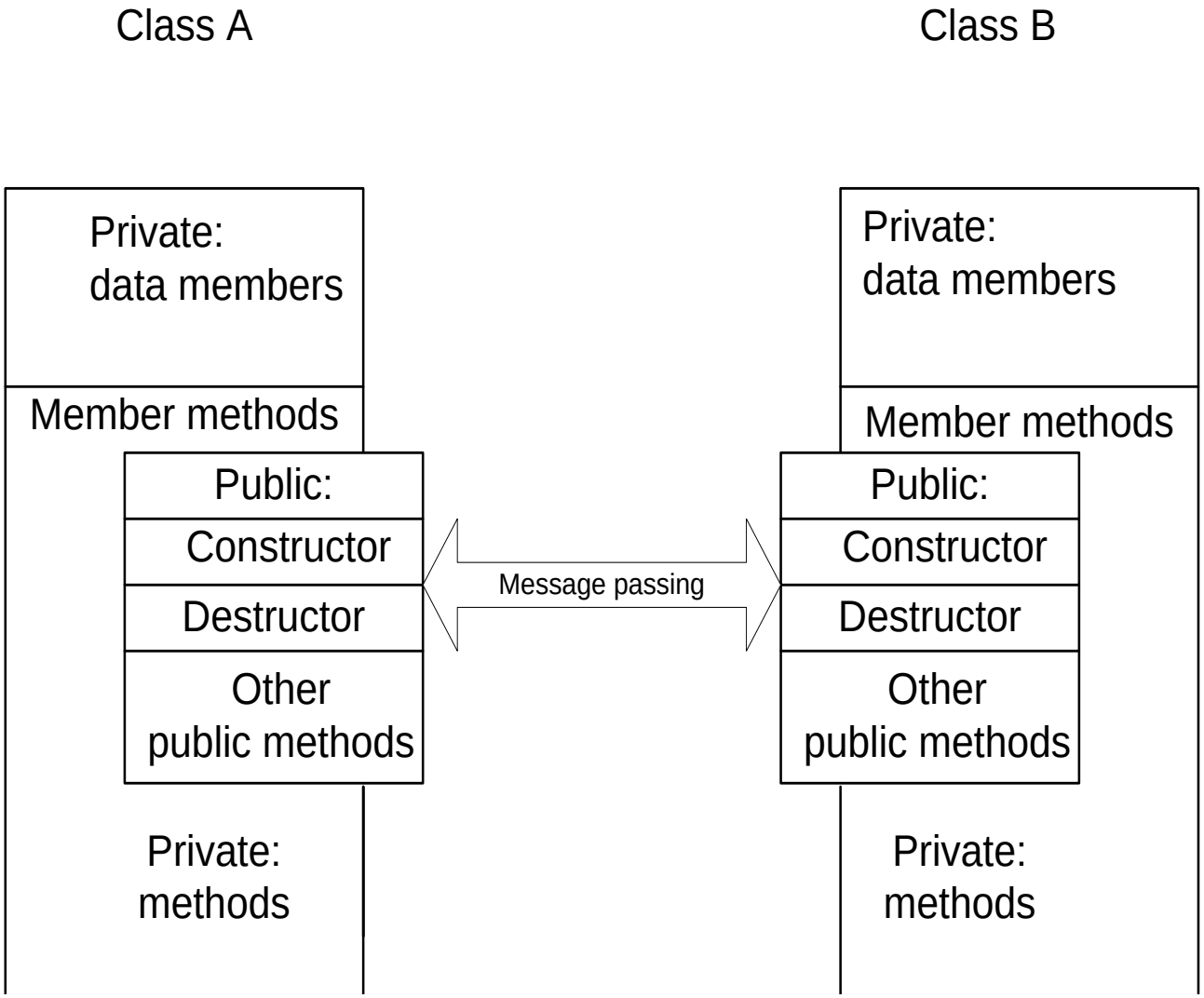
```
Str :: ~Str()
{
   delete[] pData;
};
```

Putting them Together

```
class Str
{
    char *pData;
    int nLength;
  public:
     //constructors
    Str();
    Str(char *s);
    Str(const Str &str);

     //accessors
    char* get_Data();
    int get_Len();

     //destructor
     ~Str();
};
```

```
int main()
{
     int x=3;
     Str *pStr1 = new Str("Joe");
     Str *pStr2 = new Str();
}
```

Interacting Objects



Working with Multiple Files

- **To improve the readability, maintainability and reusability, codes are organized into modules.**

- **When working with complicated codes,**
  - A set of .cpp and .h files for each class groups
    - .h file contains the prototype of the class
    - .cpp contains the definition/implementation of the class
  - A .cpp file containing main() function, should include all the corresponding .h files where the functions used in .cpp file are defined

Example : time.h

```
//   SPECIFICATION FILE                ( time .h )
//   Specifies the data members and
//   member functions prototypes.

#ifndef   _TIME_H
#define  _TIME_H

class Time
  {
    public:
          . . .

    private:
         . . .
   } ;

#endif
```

Example : time.cpp

```
//   IMPLEMENTATION FILE              ( time.cpp )
//   Implements the member functions of class Time


#include  <iostream.h>
#include  " time.h"          // also must appear in client code
    ... ...

bool  Time :: Equal (  Time otherTime ) const

//          Function value == true,    if this time equals otherTime
//                                == false , otherwise
{
    return ( (hrs == otherTime.hrs)  && (mins == otherTime.mins)
                                              && (secs  == otherTime.secs) ) ;
 }

        . . .
```

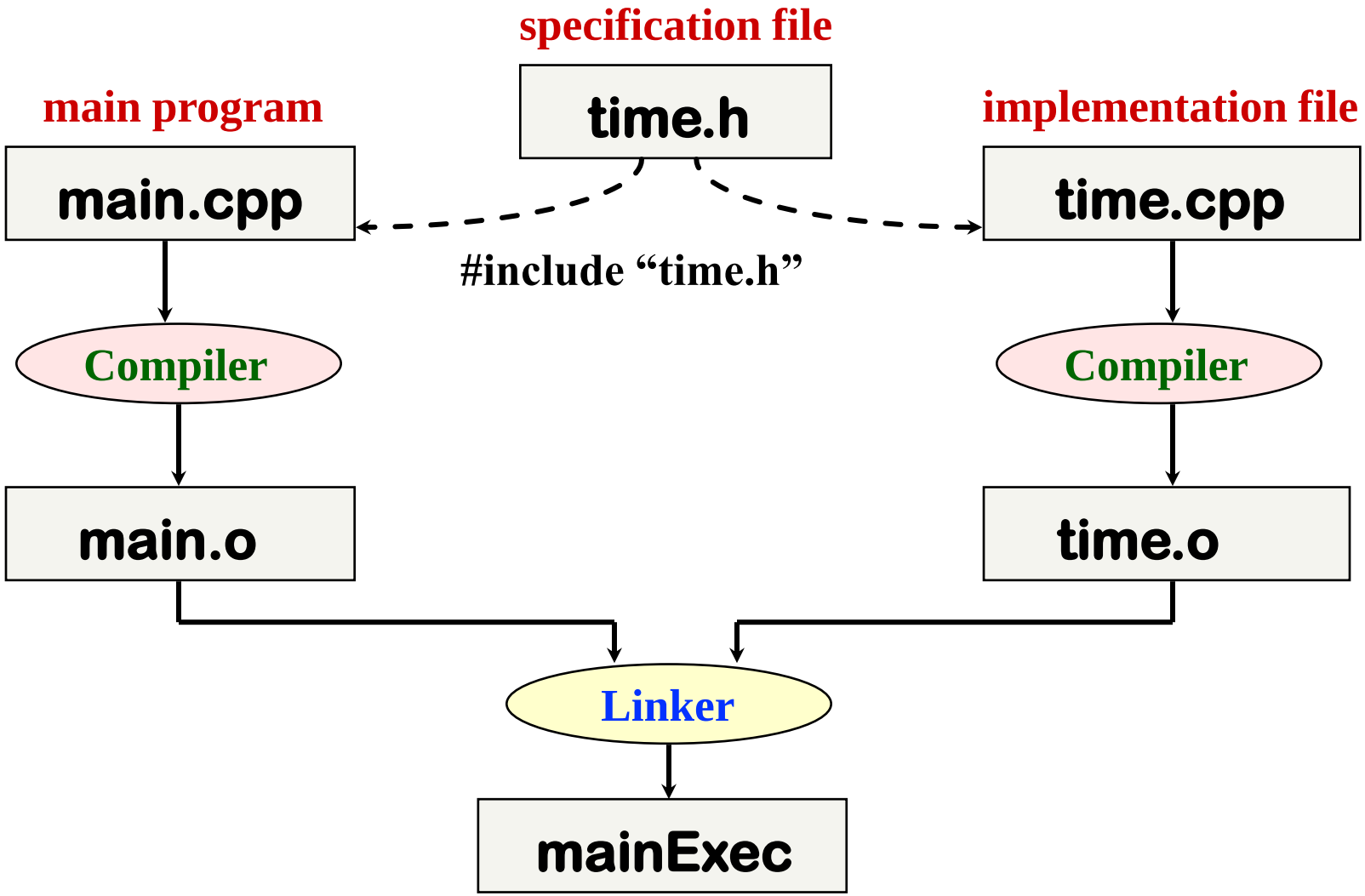
Example : main.cpp

```
//   Client Code              ( main.cpp )
#include  " time.h"

// other functions, if any

int main()
{
   … …
}
```

Compile and Run

g++ -o mainExec *main.cpp time.cpp*

Separate Compilation & Linking of Files



[Lab – Practice #1]

- **Calculator**
  - Input
    - string formula
    - Maximum 5 numbers

```
$ ./calculator
Insert Formula : 16+20*5-8/2
Result : 112
```