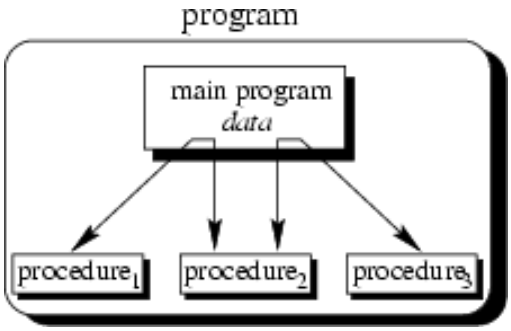
Overview of OOP and C++

Procedural Concept



- The main program coordinates calls to procedures and hands over appropriate data as parameters.

2

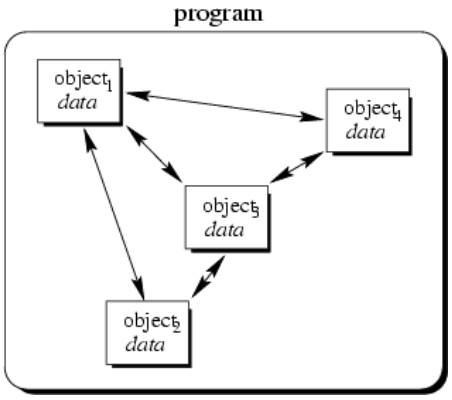
Procedural Concept

- Procedural Languages
  - C, Pascal, Basic, Fortran
  - Facilities to
    - Pass arguments to functions
    - Return values from functions
- For the rectangle problem, we develop a function

```
int compute_area (int l, int w){
        return ( l * w );
}
```

3

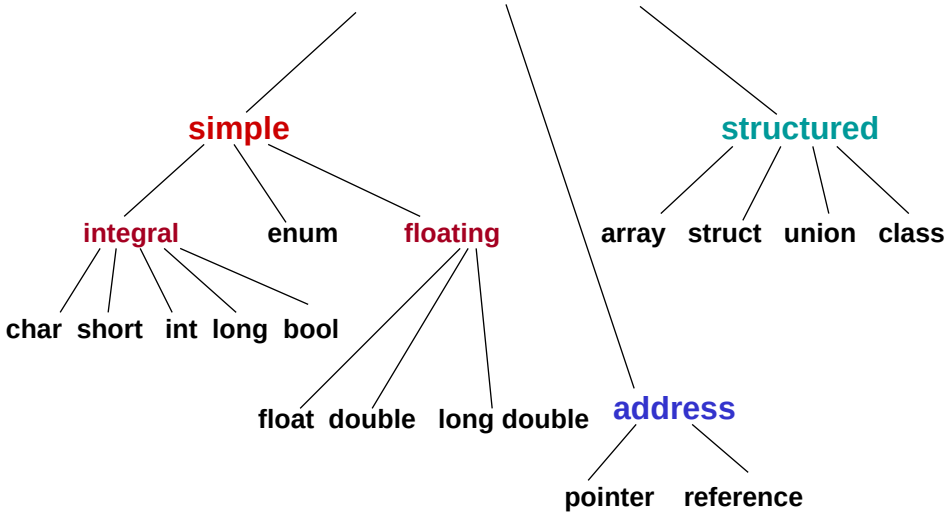
Object-Oriented Concept



- Objects of the program interact by sending messages to each other

4

C++ Data Types



5

Dynamic Memory Allocation

- *In C*, functions such as malloc() are used to dynamically allocate memory from the **Heap**.
- *In C++*, this is accomplished using the **new** and **delete** operators
- **new** is used to allocate memory during execution time
  - returns a pointer to the address where the object is to be stored
  - always returns a pointer to the type that follows the **new**

6

Operator **new** Syntax

new DataType

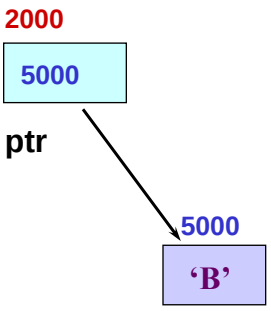
new DataType [IntExpression]

- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- The dynamically allocated object exists until the delete operator destroys it.

7

Operator new

```
char*  ptr;

ptr = new char;
*ptr = 'B';
cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

Operator **delete** Syntax

delete Pointer

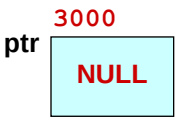
delete [ ] Pointer

- The object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.
- Good idea to set the pointer to the released memory to NULL
- Square brackets are used with delete to deallocate a dynamically allocated array.

9

Example

```
char  *ptr ;
ptr  =  new  char[ 5 ];
strcpy( ptr, "Bye" );
ptr[ 0 ] = 'u';

delete [] ptr;
ptr = NULL;
```



// deallocates the array pointed to by ptr
// ptr itself is not deallocated
// the value of ptr becomes undefined

10

Object-Oriented Programming Languages

- Characteristics of OOPL
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Overloading

11

Characteristics of OOPL

- **Encapsulation:** Combining data structure with actions
  - Data structure: represents the properties, the state, or characteristics of objects
  - Actions: permissible behaviors that are controlled through the member functions

  Data hiding: Process of making certain data inaccessible
- **Inheritance:** Ability to derive new objects from old ones
  - permits objects of a more specific class to inherit the properties (data) and behaviors (functions) of a more general/base class
  - ability to define a hierarchical relationship between objects
- **Polymorphism:** Ability for different objects to interpret functions differently

12

Characteristics of OOPL

- C++ allows function overloading
  - In C++, functions can use the same names, within the same scope, if each can be distinguished by its name *and* signature
  - The signature specifies the number, type, and order of the parameters expressed as a comma separated list of argument types

13

- declaring variables almost anywhere

```
// declare a variable when you need it
 for (int k = 1; k < 5; k++){
    cout << k;
  }
```

14

Object-Oriented Programming
Introduction to Classes

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

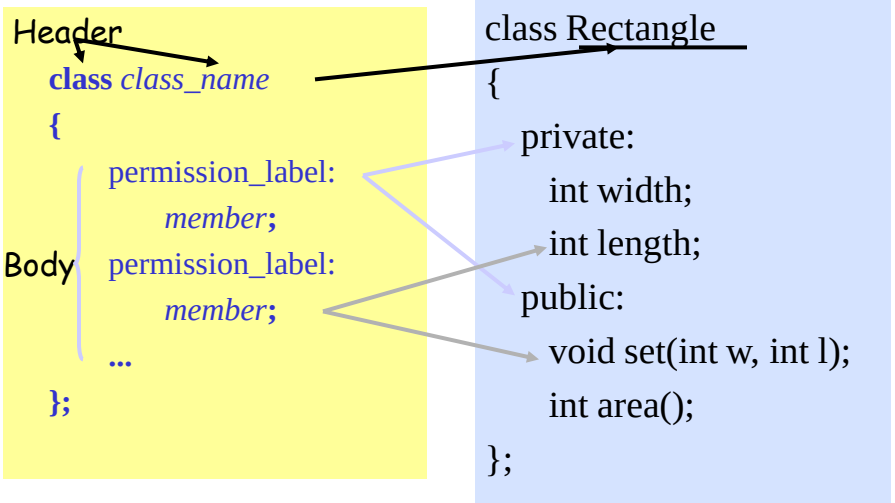
15

Class

- Class
  - A user defined type
  - Consists of both data and methods
  - Defines properties and behavior of that type
  - It makes possible encapsulation, data hiding and inheritance
- Advantages
  - Concise program
  - Code analysis easy
  - Compiler can detect illegal uses of types
- Data Abstraction
  - Separate the implementation details from its essential properties

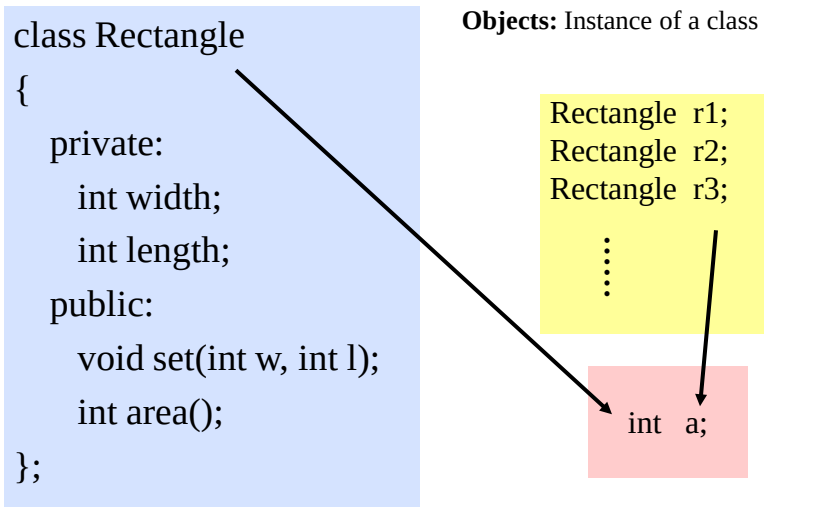
16

Define a Class Type



17

Classes & Objects



18

Class Definition
Data Members

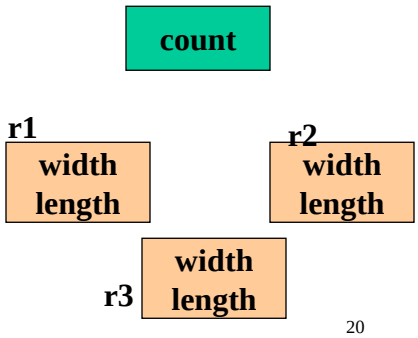
- Can be of any type, built-in or user-defined
- *non-static* data member
  - Each **class object** has its own copy
- *static* data member
  - Acts as a global variable
  - One copy per class type, e.g. counter

19

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;
```



20

Class Definition
Member Functions

- Used to
  - **access** the values of the data members
  - perform operations on the data members
- 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

21

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 nar

inline

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

scope operator

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

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

22

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

23

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;
}
```

24

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

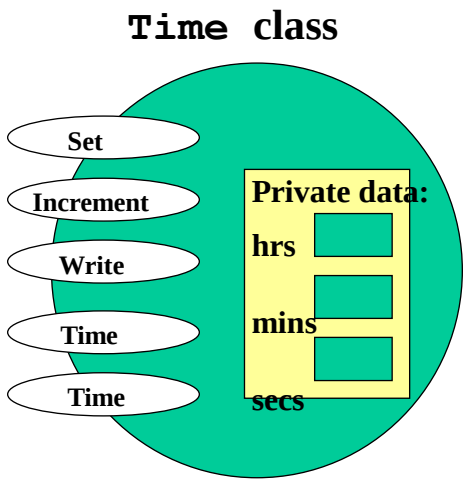
25

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 ;
} ;
```

26

Class Interface Diagram



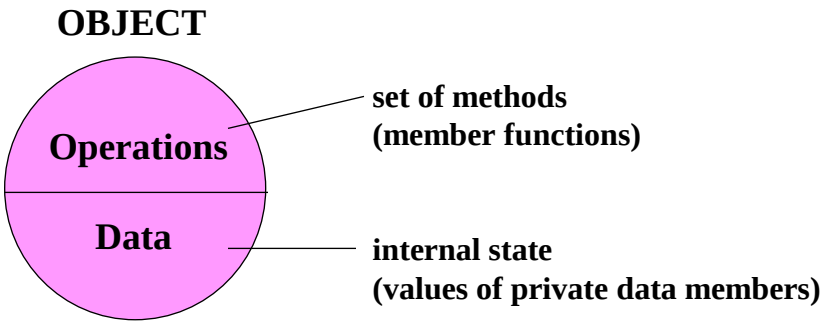
27

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

28

What is an object?



29

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;
}
```

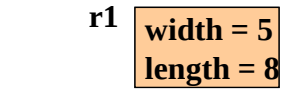
30

Declaration of an Object

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

r1 is statically allocated

```
main()
{
    Rectangle r1;
    r1.set(5, 8);
}
```



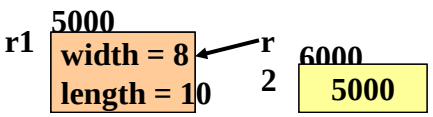
31

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 obje

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

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



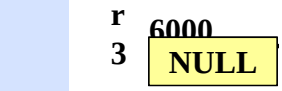
32

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;
}
```



Object Initialization

```
#include <iostream.h>

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

1. By Assignment

- 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

}
```

34

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
}
```

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 obje

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

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

r1 and r2 are both initialized by public member function set

36

Another Example

```
#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();
}
```

Constructor

Constructor in a class is a special type of subroutine called to create an object.
They have the task of initializing the object's data members and of establishing the invariant of the class, failing if the invariant is invalid. A properly written constructor leaves the resulting object in a valid state.

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

39

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);
```

41

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);
```

42

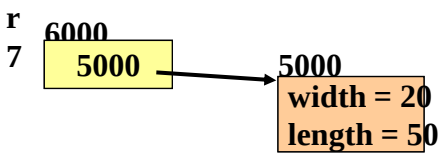
Object Initialization

```
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();
```



So far, …

- 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

45

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;  };
```

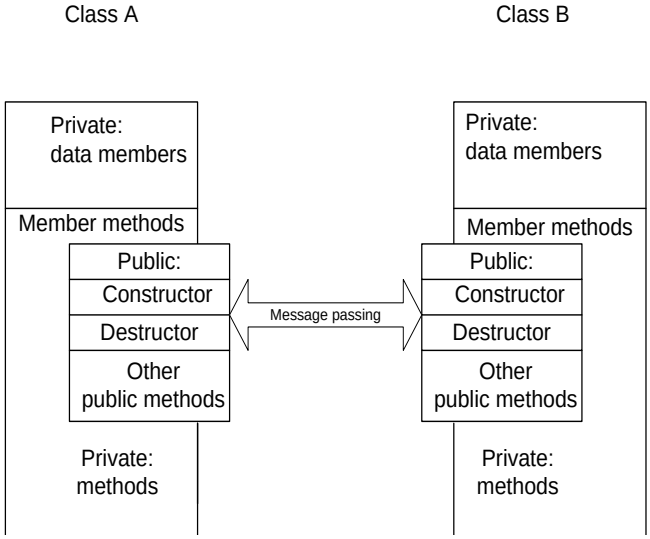
```
Str :: ~Str()
{
   delete[] pData;
};
```

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



50

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

51

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
```

52

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) ) ;
 }
     . . .
```

53

Example : main.cpp

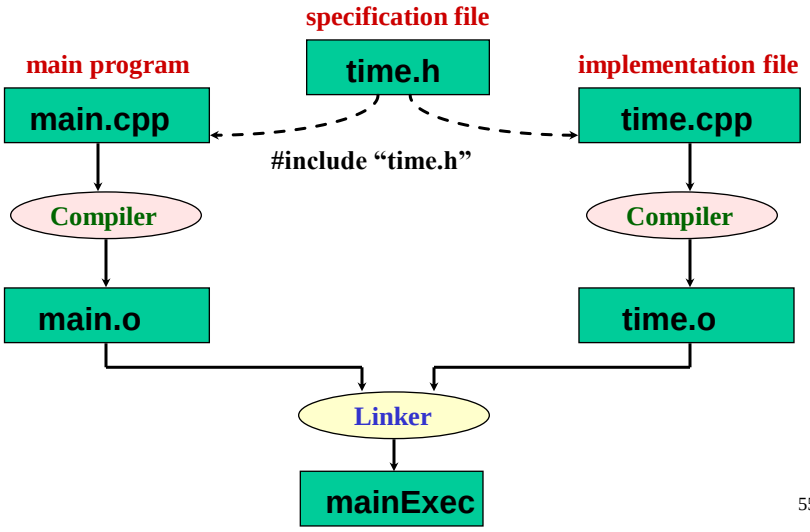
```
//   Client Code              ( main.cpp )
#include  " time.h"

// other functions, if any

int main()
{
   … …
}
```

Compile and Run

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

54

Separate Compilation and Linking of Files



55