OBJECT-ORIENTED PROGRAMMING

Lesson 4

1. Classes and Objects

Classes and Objects

1. Introduction
2. C Structures Revisited
3. Specify a Class
4. Defining Member Functions
5. A C++ Program with Class
6. Making an Outside Function Inline
7. Nesting of Member Functions
8. Private Member Functions
9. Arrays within a Class
10. Memory Allocation for Objects

Classes and Objects

11. Static Data Members
12. Static Member Functions
13. Arrays of Objects
14. Objects as Function Arguments
15. Friendly Functions
16. Returning Objects
17. const Member Functions
18. Pointers to Members
19. Local Classes

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- const member functions
- Pointers to members
- Using dereferencing operators
- Local classes

1 Introduction

- **Class** is the most important feature of C++.
- Its significance is highlighted by the fact that Stroustrup initially gave the name "**C with classes**" to his new language.

1 Introduction

- **Class**
  - A class is an extension of the idea of **structure** used in C.
  - It is a new way of creating and implementing a user-defined data type.

2 C Structures Revisited

- **Structure** in C
  - The unique features of the C language.
  - A method for packing together data of different types.
  - A convenient tool for handling a group of logically related data items.
  - A user-defined data type with a **template** that serves to define its data properties.
  - Once the **structure type** has been defined, we can create **variables** of that type using declarations that are similar to the built-in type declarations.

2 C Structures Revisited

- **Structure** in C
  - For example
    - The keyword **struct** declares **student** as a new type that hold three **fields** of different data types.

```
struct student
{
        char name[20];
        int roll_number;
        float total_marks;
};
```

    - These **fields** are known as **structure members** or **elements**.
    - The identifier **student**, which is referred to as **structure name** or **structure tag**, can be used to create variables of type student.

2 C Structures Revisited

- **Structure** in C
  - Example
    ```
    struct student A;          //C declaration
    ```
    - A is a variable of type student and has three member variables as defined by the template.
    - Member variables can be accessed using the **dot** or **period operator** as follows

    ```
    strcpy(A.name, "John");
    A.roll_number = 999;
    A.total_marks = 595.5;
    Final_total = A.total_marks + 5;
    ```

    - Structures can have **arrays**, **pointers** or **structures** as members.

2 C Structures Revisited

- Limitations of C structure
  1. The standard C does not allow the struct data type to be treated like built-in types.
     - For example
     - The complex numbers **c1**, **c2** and **c3** can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other.
     - For example `c3 = c1 + c2;` is illegal in C.

```
struct complex
{
        float x;
        float y;
};

struct complex c1, c2, c3
```

2 C Structures Revisited

- Limitations of C structure
  2. They do not permit **data hiding**.
     - Structure members can be directly accessed by the structure variables by any function anywhere in their scope.
     - In other words, the structure members are public members.

2 C Structures Revisited

- Extensions to Structure
  - C++ supports all the features of structures as defined in C.
  - But C++ has expanded its capabilities further to suit its OOP philosophy.
  - C++ attempts to bring the **user-defined** types as close as possible to the **built-in** data types.
  - And it also provides a facility to **hide** the data which is one of the main principles of OOP.
  - **Inheritance**, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

2 C Structures Revisited

- Extensions to Structure
  - In C++, a structure can have both variables and functions as members.
  - It can also declare some of its members as '**private**' so that they cannot be accessed directly by the external function.

2 C Structures Revisited

- Extensions to Structure
  - In C++, the structure name are stand-alone and can be used like any other type names.
  - In other words, the keyword struct can be omitted in the declaration of structure variables.
  - For example, we can declare the student variable A as

```
student A;          //C++ declaration
```

  - And this is an error in C.

2 C Structures Revisited

- Extensions to Structure
  - C++ incorporates all these extensions in another user-defined type known as **class**.
  - There is very little syntactical difference between **structures** and **classes** in C++ and, therefore, they can be used interchangeably with minor modifications.
  - Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the **structures** for holding only data, and **classes** to hold both the data and function.

2 C Structures Revisited

- Extensions to Structure
  - Note : The only difference between a **structure** and a **class** in C++ is that
    - By default, the members of a **class** are **private**, while, by default, the members of a **structure** are **public**.

3 Specifying a Class

- **Class**
  - It is a way to bind the **data** and its associated **functions** together.
  - It allows the data (and functions) to be **hidden**, if necessary, from external use.
  - When defining a class, we creating a new **abstract data type** that can be treated like any other built-in data type.

3 Specifying a Class

- Generally, a class specification has two parts:
  1. **Class declaration**
     - ✓ Describes the type and scope of its members.
  2. **Class function definitions**
     - ✓ Describes how the class functions are implemented.

3 Specifying a Class

- The general form of a **class declaration** is:
  - The keyword **class** specifies, that what follows is an abstract data of type **class_name**.
  - The body of a class is enclosed within braces and terminated by a semicolon.

```
class class_name
{
        private:
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declarations;
};
```

3 Specifying a Class

- Class members
  - The functions and variables declared in the class body.
  - They are usually grouped under two sections, namely, **private** and **public** to denote which of the members are **private** and which are **public**.
  - The keywords **private** and **public** are known as visibility labels.
  - Note that theses keywords are followed by a **colon**.

3 Specifying a Class

- **Private** & **public**
  - **Private**
    - The class members that have been declared as private can be accessed only from within the class.
  - **Public**
    - Public members can be accessed from outside the class also.
  - The **data hiding** (using private declaration) is the key feature of OOP.
  - The use of the keyword **private** is optional.
  - By default, the members of a class are **private**.

3 Specifying a Class

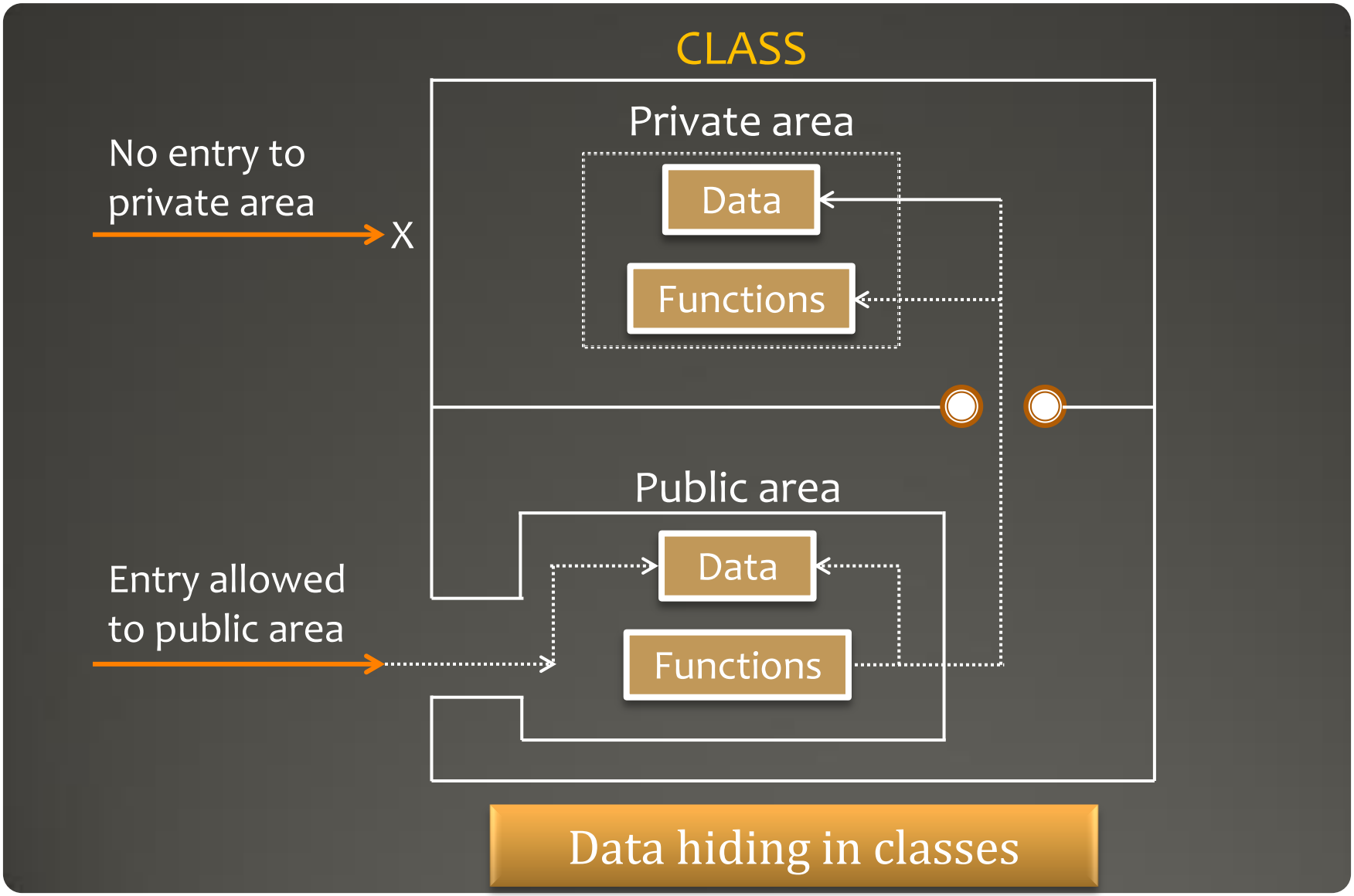
- **Private** & **public**
  - If both the labels are missing, then, be default, all the members are **private**.
  - Such a class is completely hidden from the outside world and does not serve any purpose.

3 Specifying a Class

- Data members & member functions
  - **Data members**
    - The variables declared inside the class.
  - **Member function**
    - The functions declared inside the class.
  - Only the **member functions** can have access to the **private** data members and private functions.
  - The **public** members (both functions and data) can be accessed from outside the class.

3 Specifying a Class

- **Encapsulation**
  - The binding of data and functions together into a single class.



Data hiding in classes

3 Specifying a Class

- A Simple Class Example
  - A typical class declaration would look like:

```
class item
{
        int number;                          //variables declaration
        float cost;                          //private be default
  public:
        void getdata(int a, float b);        //functions declaration
        void putdata(void);                  //using prototype
};
```

3 Specifying a Class

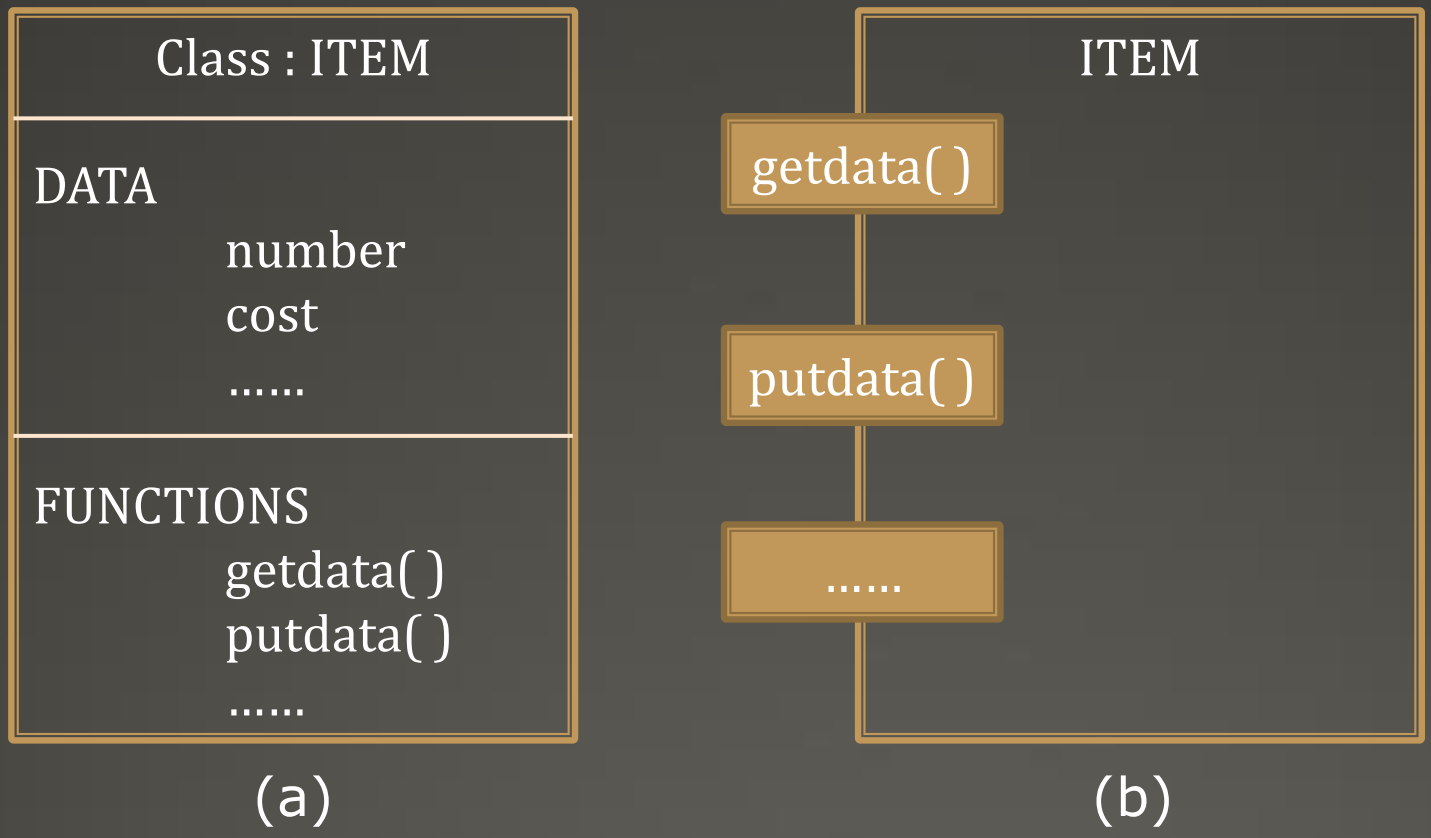
- A Simple Class Example
  - We usually give a class some meaningful name.
  - The name of a class will become a new type identifier that can be used to declare **instances** of that class type.
  - The class item contains two **data members** and two **function members**.
  - The data members are **private** by default while both the functions are **public** by declaration.

3 Specifying a Class

- A Simple Class Example
  - The function **getdata()** can be used to assign values to the member variables number and cost and **putdata()** for displaying their values.
  - These functions provide the only access to the data members from outside the class.

3 Specifying a Class

- A Simple Class Example
  - Note that the functions are declared, not defined.
  - Actual function definitions will appear later in the program.
  - The **data members** are usually declared as **private** and the **member functions** as **public**.



Two different notations of a class

3 Specifying a Class

- **Creating objects**
  - The declaration of **item** does not define any objects of **item** but only specifies **what** they will contain.
  - Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable).
  - For example
```
item  x;  //memory for x is created
```
    creates a variable x of type **item**.
  - In C++, the class variables are known as **objects**.
  - Therefore, **x** is called an object of type **item**.

3 Specifying a Class

- **Creating objects**
  - We can also declare more than one object in one statement.
  - Example

```
item  x, y, z;
```

3 Specifying a Class

- **Creating objects**
  - The declaration of an object is similar to that of a variable of any basic type.
  - The necessary memory space is allocated to an object at this stage.
  - Note that class specification, like a structure, provides only a **template** and does not create any memory space for the objects.

3 Specifying a Class

- **Creating objects**
  - Objects can be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures.
  - So, the definition

```
class  item
{
            ......
} x, y, z;
```

would create the objects **x**, **y** and **z** of type **item**.

  - This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

3 Specifying a Class

- **Accessing Class Members**
  - The **private** data of a class can be accessed only through the member functions of that class.
  - The main() cannot contain statements that access data members directly.

3 Specifying a Class

- **Accessing Class Members**
  - The following is the format for calling a **member function**

```
object-name.function-name (actual-arguments);
```

  - For example, the function call statement `x.getdata(100, 75.5);` is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object x by implementing the **getdata()** function.
  - The assignments occur in the actual function.
  - Similarly, the statement `x.putdata( );` would display the values of data members.

3 Specifying a Class

- **Accessing Class Members**
  - Note that a member function can be invoked only by using an object (of the same class).
  - The statement like `getdata(100, 75.5);` has no meaning.
  - Similarly, the statement `x.number = 100;` is also illegal.
  - Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

3 Specifying a Class

- **Accessing Class Members**
  - Object communicate by sending and receiving messages.
  - This is achieved through the member functions.
  - For example `x.putdata( );` sends a message to the object **x** requesting it to display its contents.

3 Specifying a Class

- **Accessing Class Members**
  - A variable declared as **public** can be accessed by the objects directly.
  - Note: The use of data in this manner defeats the very idea of **data hiding** and therefore should be avoided.

```
class  xyz
{
            int  x;
            int  y;
  public:
            int  z;
};
……
……
xyz  p;
p.x = 0;              //error,  x is private
p.z = 10;             //OK, z is public
……
```

4 Defining Member Functions

- Member functions can be defined in two places:
  - Outside the class definition
  - Inside the class definition

4 Defining Member Functions

1. Outside the class definition
   - Member functions that are declared inside a class have to be defined separately outside the class.
   - Their definitions are very much like the normal functions.
   - They should have a function header and a function body.
   - Since C++ does not support the old version of function definition, the ANSI **prototype** form must be used for defining the function header.

4 Defining Member Functions

1. Outside the class definition
   - An important difference between a member function and a normal function is that
     - A member function incorporates a membership ‘identity label’ in the header.
   - This ‘label’ tells the compiler which **class** the function belongs to.

4 Defining Member Functions

1. Outside the class definition
   - The general form of a member function definition is:

```
return-type  class-name :: function-name (argument  declaration)
{
        Function body
}
```

   - The membership label class-name :: tells the compiler that the function **function-name** belongs to the class **class-name**.
   - That is, the scope of the function is restricted to the **class-name** specified in the header line.
   - The symbol :: is called the **scope resolution** operator.

4 Defining Member Functions

1. Outside the class definition
   - For instance, the member functions **getdata()** and **putdata()**
   - Since these functions do not return any value, their return-type is void.
   - Function arguments are declared using the ANSI prototype.

```
void  item :: getdata (int  a, float  b)
{
        number = a;
        cost = b;
}
```

```
void  item :: putdata (void)
{
        cout << "Number :" << number << "\n";
        cout << "Cost :" << cost << "\n";
}
```

4 Defining Member Functions

1. Outside the class definition
   - Some characteristics the member functions have that are often used in the program development:
     - Several different classes can use the same function name. The '**membership label**' will resolve their scope.
     - Member functions can access the **private** data of the class. A non-member function cannot do so. (However an exception to this rule is a **friend function**)
     - A member function can call another member function directly, without using the dot operator.

4 Defining Member Functions

2. Inside the class definition
   - Replace the function declaration by the actual function definition inside the class.

4 Defining Member Functions

2. Inside the class definition
   - For example

```
class item
{
        int  number;
        float  cost;
  public:
        void  getdata(int  a,  float  b);       //declaration
                // inline function
        void  putdata(void);                    //definition inside the class
        {
                cout << number << "\n";
                cout << cost << "\n";
        }
};
```

4 Defining Member Functions

2. Inside the class definition
   - When a function is defined inside a class, it is treated as an **inline function**.
   - Therefore, all the restrictions and limitations that apply to an inline function are also applicable here.
   - Normally, only small functions are defined inside the class definition.

5 A C++ Program with Class

- Example (Program 5.1)
  - Note the use of statements such as `number = a;` in the function definition of getdata().
    - This show that the member functions can have direct access to private data items.
  - The program creates two objects, x and y in two different statements, which can be combined in one statement.

```
item  x, y;          //creates a list of objects
```

6 Making an Outside Function Inline

- One of the objectives of OOP is to separate the details of implementation from the class definition.
- It is therefore good practice to define the member functions outside the class.

6 Making an Outside Function Inline

- We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition.

6 Making an Outside Function Inline

- Example

```
class item
{
        ……
  public:
        void  getdata(int  a,  float  b);          //declaration
};
inline  void  item :: getdata (int  a, float  b);          //definition
{
        number = a;
        cost = b;
}
```

7 Nesting of Member Functions

- A member function of a class can be called only by an object of that class using a dot operator.
- However, there is an exception to this.
- Nesting of member functions
  - A member function can be called by using its name inside another member function of the same class.
- Example (Program 5.2)

8 Private Member Functions

- Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from outside calls.
  - Tasks such as deleting an account in a customer file, or providing increment to an employee are events serious consequences.
  - And therefore the functions handling such tasks should have restricted access.
  - We can place these functions in the private section.

8 Private Member Functions

- A private member function can only be called by another function that is a member of its class.
- Even an object cannot invoke a private function using the dot operator.

8 Private Member Functions

- Consider a class as defined below

```
class  sample
{
            int  m;
            void  read(void);          //private member function
  public:
            void  update(void);
            void  write(void);
};
```

  - If **s1** is an object of **sample**, then the statement is illegal.

```
s1.read( );//won't work; objects cannot access private members
```

8 Private Member Functions

- Consider a class as defined below

```
class  sample
{
            int  m;
            void  read(void);          //private member function
  public:
            void  update(void);
            void  write(void);
};
```

  - However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void  sample :: update (void)
{
            read( );              //simple call; no object used
}
```

9 Arrays within a Class

- The arrays can be used as member variables in a class.
- The following definition is valid.

```
const  int  size = 10;                    //provides value for array size

class  array
{
        int   a[size];                    //'a' is int type array
public:
        void  setval(void);
        void  display(void);
};
```

9 Arrays within a Class

- The arrays variable declared as a member of the class can be used in the member functions, like any other array variable.
- Similarly, we may use other member functions to perform any other operations on the array values.

9 Arrays within a Class

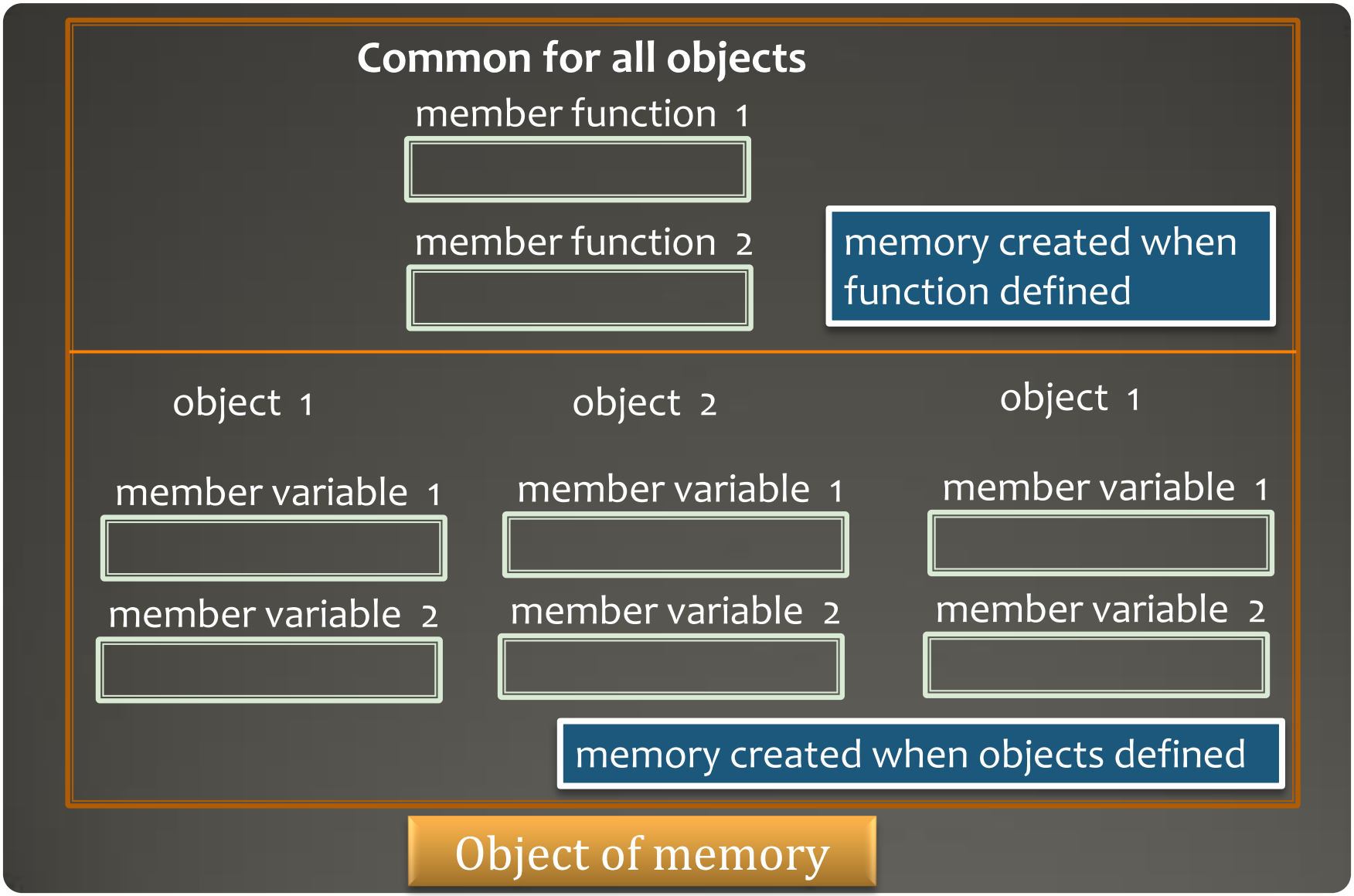
- Example
  - A shopping list of items for which we place an order with a dealer every month.
  - The list includes details such as the code number and price of each item.
  - Operations such as adding an item to the list, deleting an item from the list and printing the total value of the order.
  - Program 5.3
    - The program implements all the tasks using a menu-based user interface.

10 Memory Allocation for Objects

- The memory space for objects is allocated when they are declared and not when the class is specified.
- Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification.

10 Memory Allocation for Objects

- Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.
- Only space for member variables is allocated separately for each object.
- Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.


Common for all objects
member function 1
member function 2
memory created when function defined
object 1
object 2
object 1
member variable 1
member variable 2
member variable 1
member variable 2
member variable 1
member variable 2
memory created when objects defined
Object of memory

11 Static Data Members

- A data member of a class can be qualified as static.
- The properties of a static member variable are similar to that of a C static variable.

11 Static Data Members

- Special characteristics of a static member variable
  - It is initialized to zero when the first object of its class is created.
    No other initialization is permitted.
  - Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
  - It is visible only within the class, but its lifetime is the entire program.

11 Static Data Members

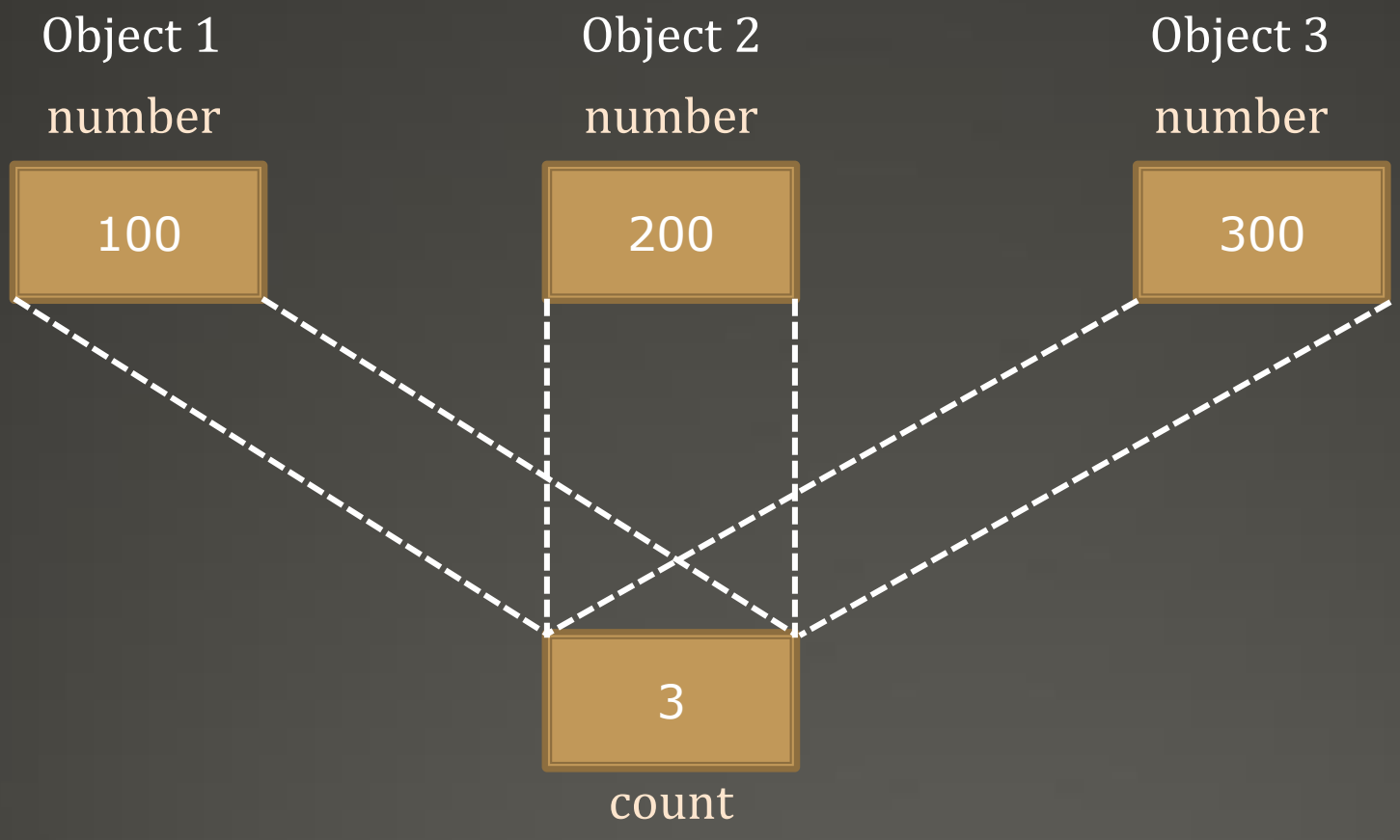
- Static variables are normally used to maintain values common to the entire class.
  - For instance, a static data member can be used as a counter that records the occurrences of all the objects.
  - Example (Program 5.4)

11 Static Data Members

- Notice the following statement in the program

```
int item :: count;          //definition of static data member
```

  - The type and scope of each static member variable must be defined outside the class definition.
  - This is necessary because the static data members are stored separately rather than as a part of an object.
  - Since the static data members are associated with the class itself rather than with any class object, they are also known as class variables.



Sharing of a static data member

11 Static Data Members

- While defining a static variable, some initial value can be assigned to the variable.
  - For instance, the following definition gives count the initial value 10.

```
int item::count = 10;
```

12 Static Member Functions

- A member function that is declared static has the following properties
  - A static function can have access to only other **static members** (functions or variables) declared in the same class.
  - A static function can be called using the class name (instead of its object) as follows

```
class-name : : function-name;
```

  - Example (Program 5.5)

12 Static Member Functions

- Note
  - The statement `code = ++count` is executed whenever **setcode()** function is invoked and the current value of **count** is assigned to **code**.
  - Since each object has its own copy of **code**, the value contained in **code** represents a unique number of its object.

12 Static Member Functions

- Remember
  - The following function definition will not work

```
static  void  showcount( )
{
        cout << code;      //code is not static
}
```

13 Arrays of Objects

- An array can be of any data type including struct.
- Similarly, we can also have arrays of variables (called arrays of objects) that are of the type class.

13 Arrays of Objects

- Consider the following class definition

```
class  employee
{
        char  name[30];
        float  age;
public:
        void  getdata (void);
        void  putdata (void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees.

Example

```
employee  manager[3];    //array of manager
employee  foreman[15];   //array of foreman
employee  worker[75];    //array of worker
```

13 Arrays of Objects

- The array manager contains three objects (managers), namely, manager[0], manager[1] and manager[2], of type employee class.
- Similarly, the foreman array contains 15 objects (foremen) and the worker array contains 75 objects (workers).

13 Arrays of Objects

- Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions.
  - For example, the statement `manager[i].putdata( );` will display the data of the ith element of the array **manager**.
  - That is, the statement requests the object **manager[i]** to invoke the member function **putdata()**.

13 Arrays of Objects

- An array of objects is stored inside the memory in the same way as a multi-dimensional array.

Storage of data items of an object array

13 Arrays of Objects

- Note
  - Only the space for data items of the objects is created.
  - Member functions are stored separately and will be used by all the objects.
- Example (Program 5.6)

14 Objects as Function Arguments

- An object may be used as a function argument, which can be done in two ways.
  1. Pass-by-value : A copy of the entire object is passed to the function.
     - Since a copy of the object is passes to the function, any changes made to the object inside the function do not affect the object used to call the function.

14 Objects as Function Arguments

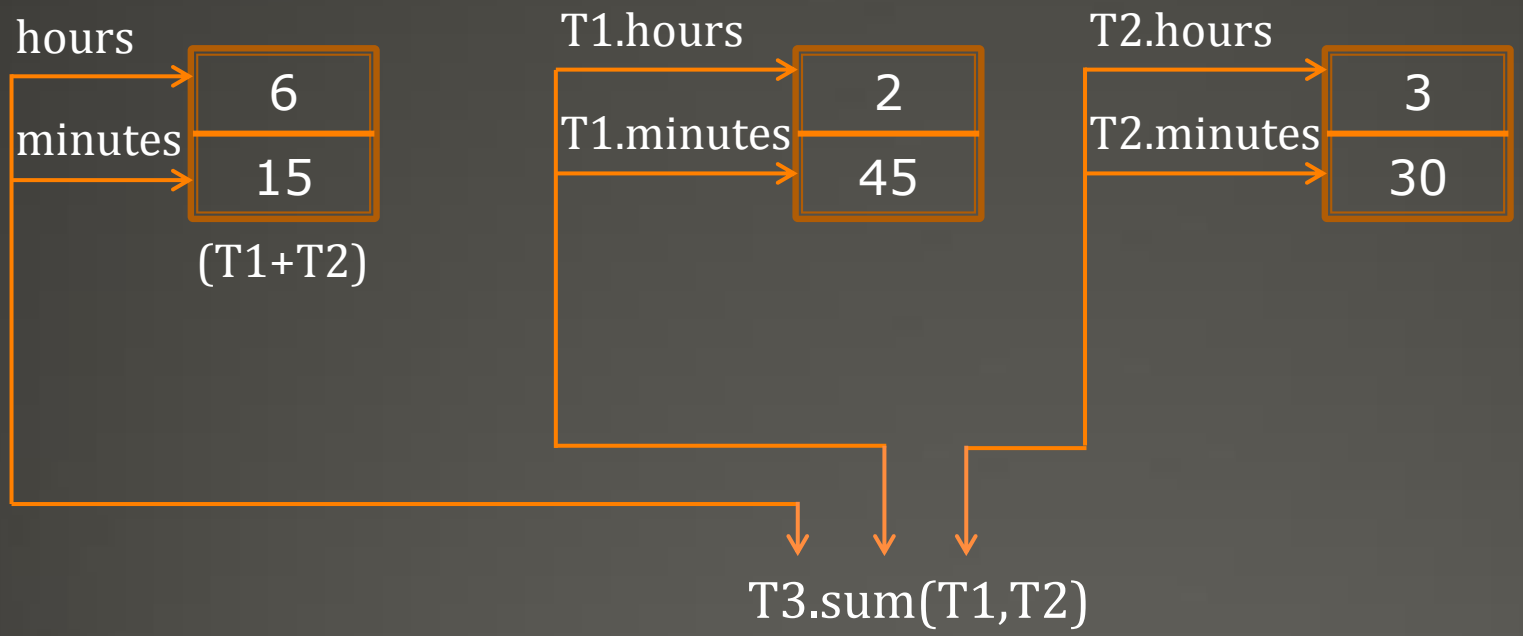
- An object may be used as a function argument, which can be done in two ways.
  2. Pass-by-reference : Only the address of the object is transferred to the function.
     - When an address of the object is passed, the called function works directly on the actual object used in the call.
     - This means that any changes made to the object inside the function will reflect in the actual object.
     - This method is more efficient since it requires to pass only the address of the object and not the entire object.

14 Objects as Function Arguments

- Example (Program 5.7)
- Note
  - Since the member function sum() is invoked by the object T3, with the objects T1 and T2 as arguments, it can directly access the hours and minutes variables of T3.
  - But the members of T1 and T2 can be accessed only by using the dot operator(like T1.hours and T1.minutes).
  - Therefore, inside the function sum(), the variables hours and minutes refer to T3, T1.hours and T1.minutes refer to T1, and T2.hours and T2.minutes refer to T2.

14 Objects as Function Arguments

- The figure below illustrates how the members are accessed inside the function sum().



Accessing members of objects within a called function

14 Objects as Function Arguments

- An object can also be passed as an argument to a non-member function.
  - However, such functions can have access to the public member functions only through the objects passed as arguments to it.
  - These functions cannot have access to the private data members.

15 Friendly Functions

- The private members cannot be accessed from outside the class.
  - That is, a non-member function cannot have an access to the private data of a class.

15 Friendly Functions

- However, there could be a situation where we would like two classes to share a particular function.
  - For example, consider a case where two classes, **manager** and **scientist**, have been defined.
  - We would like to use a function **income_tax()** to operate on the objects of both these classes.
  - In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes.
  - Such a function need not be a member of any of these classes.

15 Friendly Functions

- To make an outside function "friendly" to a class, we have to declare this function as friend of the class
  - The function declaration should be preceded by the keyword friend.
  - The function is defined elsewhere in the program like a normal C++ function.
  - The function definition does not use either the keyword friend or the scope operator ::.

```
class  ABC
{
            ……
public:
            ……
            friend  void  xyz(void);    //declaration
}
```

15 Friendly Functions

- A function can be declared as a friend in any number of classes.
- A friend function, although not a member function, has full access rights to the private members of the class.

15 Friendly Functions

- A friend function possesses certain special characteristics
  - It is not in the scope of the class to which it has been declared as friend.
  - Since it is not in the scope of the class, it cannot be called using the object of that class.
  - It can be invoked like a normal function without the help of any object.

15 Friendly Functions

- A friend function possesses certain special characteristics
  - Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name(e.g. A.x).
  - It can be declared either in the public or the private part of a class without affecting its meaning.
  - Usually, it has the objects as arguments.
- The friend function are often used in operator overloading.

15 Friendly Functions

- Example (Program 5.8)
- Note
  - The friend function accesses the class variables a and b by using the dot operator and the object passed to it.
  - The function call mean(X) passes the object X by value to the friend.

15 Friendly Functions

- Member functions of one class can be friend functions of another class.
  - In such cases, they are defined using the scope resolution operator.

15 Friendly Functions

```
class  X
{
        ……
        int  fun1( );                    //member function of X
        ……
};
Class Y
{
        ……
        friend  int  X : : fun1( );      //fun1( ) of X is friend of Y
        ……
};
```

- The function fun1() is a member of class X and a friend of class Y.

15 Friendly Functions

- We can also declare all the member functions of one class as the friend functions of another class.
  - In such cases, the class is called a friend class.

```
class Z
{
        ……
        friend class X;   //all member functions of X are friends to Z
};
```

15 Friendly Functions

- Example (Program 5.9)
  - Demonstrates how friend functions work as a bridge between the classes.
- Note
  - The function max() has arguments from both XYZ and ABC.
  - When the function max() is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as `class ABC;` which is known as '**forward**' declaration.

15 Friendly Functions

- Call-by-reference
  - A friend function can be called by reference.
  - In this case, local copies of the objects are not made.
  - Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

15 Friendly Functions

- Call-by-reference
  - This method can be used to alter the values of the private members of a class.
  - Remember, altering the values of private members is against the basic principles of data hiding.
  - It should be used only when absolutely necessary.

15 Friendly Functions

- Call-by-reference
  - Example (Program 5.10)
    - Shows how to use a common friend function to exchange the private values of two classes.
    - The function is called by reference.

15 Friendly Functions

- Call-by-reference
  - Example (Program 5.10)
    - The objects **x** and **y** are aliases of **C1** and **C2** respectively.
    - The following statements directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

```
int  temp = x.value1;
x.value1 = y.value2;
y.value2 = temp;
```

16 Returning Objects

- A function cannot only receive objects as arguments but also can return them.
- Example (Program 5.11)
  - The program adds two complex numbers **A** and **B** to produce a third complex number **C** and displays all the three numbers.
  - Illustrates how an object can be created (within a function) and returned to another function.

17 Const Member Functions

- If a member function does not alter any data in the class, we may declare it as a **const** member function

```
void  mul(int, int)  const;
double  get_balance( )  const;
```

  - The qualifier **const** is appended to the function prototypes (in both declaration and definition).
  - The compiler will generate an error message if such functions try to alter the data values.

18 Pointer to Members

- It is possible to take the address of a member of a class and assign it to a pointer.
- The address of a member can be obtained by applying the operator **&** to a "fully qualified" class member name.
- A class member pointer can be declared using the operator **::*** with the class name.

18 Pointer to Members

- For example, given the class

```
class A
{
        private :
                int m;
        public:
                void show( );
};
```

  - We can define a pointer to the member

```
int A ::* ip = &A :: m;
```

  - The **ip** pointer created thus acts like a class member in that it must be invoked with a class object.
  - The phrase **A::*** means "pointer-to-member of A class".
  - The phrase **&A::m** means "address of the m member of A class".
  - The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions).

18 Pointer to Members

- Remember, the following statement is not valid

```
int  * ip = &m;
```

  - This is because **m** is not simply an int type data.
  - It has meaning only when it is associated with the class to which it belongs to.
  - The scope operator must be applied to both the pointer and the member.

18 Pointer to Members

- Assume that **a** is an object of **A** declared in a member function.
  - We can access **m** using the pointer **ip** as

```
cout << a.*p;      //display
cout << a.m;       //same as above
```

  - The code

```
ap = &a;                  //ap is pointer to object a
cout << ap -> *ip;        //display  m
cout << ap -> m;          //same as above
```

    - The **dereferencing operator ->*** is used to access a member when we use pointers to both the object and the member.
    - The **dereferencing operator .*** is used when the object itself is used with the member pointer.
    - Note that ***ip** is used like a member name.

18 Pointer to Members

- We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the main

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

  - The precedence of **()** is higher than that of **.*** and **->***, so the parentheses are necessary.

18 Pointer to Members

- Example (Program 5.12)
  - Illustrates the use of dereferencing operators to access the class members.

19 Local Classes

- Local classes
  - Classes that defined and used inside a function or a block.
  - Example

```
void  test(int  a)              //function
{
        ……
        class student           //local class
        {
                ……              //class definition
        };
        ……
        student  s1(a);         //create student object use student object
}
```

19 Local Classes

- Local classes can use global variables and static variables declared inside the function but cannot use automatic local variables.
- The global variables should be used with the scope operator (::).

19 Local Classes

- Some restrictions in constructing local classes.
  - They cannot have static data members and member functions must be defined inside the local classes.
  - Enclosing function cannot access the private members of a local class.
  - However, we can achieve this by declaring the enclosing function as a friend.