OBJECT-ORIENTED PROGRAMMING

Lesson 3

1. Functions in C++
2. Classes and Objects

Functions in C++

1. Introduction
2. The Main Function
3. Function Prototyping
4. Call by Reference
5. Return by Reference
6. Inline Functions
7. Default Arguments
8. const Arguments
9. Function Overloading
10. Friend and Virtual Functions

Key Concepts

- Return type in main()
- Function prototyping
- Call by reference
- Call by value
- Return by reference
- Inline functions
- Default arguments
- Const arguments
- Function overloading

1 Introduction

- Dividing a program into functions is one of the major principles of top-down, structured programming.
- It is possible to reduce the size of a program by calling and using functions at different places in the program.

1 Introduction

- A syntax similar to the following in developing C programs.

```
void  show( );                  /* Function declaration */
main ( )
{
        ……
        show( );                /* Function call */
        ……
}
void  show( )                   /* Function definition */
{
        ……
        ……                      /* Function body */
        ……
}
```

1 Introduction

- Functions continue to be the building blocks of C++ programs.
- Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the argument passed to it.
- Most of these modifications are aimed at meeting the requirements of object-oriented facilitiles.

2 The Main Function

- In C++, the main() returns a value of type **int** to the operating system.
- C++ explicit defines main() as matching one of the following prototypes:

```
int main( );
int main(int argc, char * argv[ ]);
```

- The functions that have a return value should use the **return** statement for termination.
- The main() functions in C++ is defined as:

```
int main( );
{
        ……
        return 0;
}
```

3 Function Prototyping

- Function prototyping
  - One of the major improvements to C++ functions.
  - Describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

3 Function Prototyping

- Function prototyping
  - A **template** is always used when declaring and defining a function.
  - When a function is called, the compiler uses the **template** to ensure that proper arguments are passed, and the return value is treated correctly.
  - Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.
  - These checks and controls did not exist in the conventional C functions.

3 Function Prototyping

- A major difference in prototyping between C and C++
  - When C++ make the prototyping **essential**, ANSI C makes it **optional** (perhaps, to preserve the compatibility with the classic C)

3 Function Prototyping

- Function prototype
  - It is a declaration statement in the calling program.
  - It is of the form `type function-name (argument-list);`
  - The **argument-list** contains the **types** and **names** of arguments that must be passed to the function.
  - Example `float volume (int x, float y, float z);`

3 Function Prototyping

- Function prototype
  - Note that each argument variable must be declared independently inside the parentheses.
  - A combined declaration `float volume (int x, float y, z);` is illegal.

3 Function Prototyping

- Function prototype
  - In a function **declaration**, the names of the arguments are dummy variables and therefore, they are optional.
    - That is, the form `float volume (int, float, float);` is acceptable at the place of declaration.

3 Function Prototyping

- Function prototype
  - In the function **definition**, names are required because the arguments must be referenced inside the function.
    - Example

```
float volume (int a, float b, float c);
{
        float v = a * b * c;
        ……
}
```

    - The function volume() can be invoked in a program as follows

```
float cube1 = volume (b1, w1, h1);          //Function call
```

    - The variable b1, w1, and h1 are known as the **actual parameters** which specify the dimensions of cube1.

3 Function Prototyping

- Function prototype
  - We can also declare a function with an **empty argument list**
    - Example `void display( );`
  - In C++, it means that the function does not pass any parameters.
  - It is identical to the statement `void display(void);`

3 Function Prototyping

- Example (eg2-9)

4 Call by Reference

- **Call-by value**
  - In traditional C, a function call passes arguments by **value**.
  - This mechanism is fine if the function does not need to alter the values of the original variables in the calling program.

4 Call by Reference

- **Call-by reference**
  - There may arise situations where we would like to change the values of variables in the calling program.
    - For example, if a function is used for **bubble sort**, then it should be able to alter the values of variables in the calling function, which is not possible if the **call-by-value** method is used.

4 Call by Reference

- **Call-by reference**
  - Provisions of the **reference** variables in C++ permits us to pass arguments to the functions by reference.
  - When we pass arguments by reference, the '**formal**' arguments in the called function become aliases to the '**actual**' arguments in the calling function.
  - This means that when the function is working with its own arguments, it is actually working on the original data.

4 Call by Reference

- **Call-by reference**
  - Consider the following function

```
void  swap (int  &a, int  &b)        //a and b are reference variables
{
        int  t = a;                  //Dynamic initialization
        a = b;
        b = t;
}
```

  - Now, if **m** and **n** are two integer variables, then the function call `swap (m, n);` will exchange the values of **m** and **n** using their aliases (reference variables) **a** and **b**.

4 Call by Reference

- **Call-by reference**
  - In traditional C, it is accomplished using **pointers** and **indirection**.

```
void  swap1 (int  &a, int  &b)        /* Function definition */
{
        int  t;
        t = *a;                 /* Assign the value at address a to t */
        *a = *b;                /* Put the value at b into a */
        *b = t;                 /* Put the value at t into b */
}
```

  - The function can be called as follows

```
swap1 (&x, &y);              /* Call by passing address of variables */
```

5 Return by Reference

- A function can also return a reference.
  - Consider the following function:

```
int & max(int &x, int &y)
{
        if (x>y)
                return x;
        else
                return y;
}
```

- Since the return type of max( ) is int &, the function returns reference to x or y (and not the values).
- Then a function call such as `max (a, b)` will yield a reference to either a or b depending on their values.
- This means that this function call can appear on the left-hand side of an assignment statement.
- The statement `max (a, b) = -1` is legal and assigns -1 to a if it is larger, otherwise -1 to b.

5 Return by Reference

- Example (Eg2-13)

6 Inline Functions

- **Problem** of using functions
  - One of the objectives of using functions is to save some memory space, which becomes appreciable when a function is likely to be called many times.
  - However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the calling function, saving registers, pushing arguments into the stack, and returning to the calling function.
  - When a function is small, a substantial percentage of execution time may be spent in such overheads.

6 Inline Functions

- Solution of C
  - Use macro definitions, popularly known as **macros**.
  - The major drawback with **macros** is that they are not really functions and therefore, the usual error checking does not occur during compilation.

6 Inline Functions

- Solution of C++
  - To eliminate the cost of calls to small functions, C++ proposes a new feature called ***inline function***.
  - An ***inline function*** is a function that is expanded in line when it is invoked.
  - And the compiler replaces the function call with the corresponding function code (something similar to macros expansion).

6 Inline Functions

- Inline functions
  - They are defined as
  - Examples

```
inline  function - header
{
        function body
}
```

```
inline  double cube (double  a)
{
        return (a*a*a);
}
```

  - The above inline function can be invoked by statements like

```
c = cube (3.0);
d = cube (2.5 + 1.5);
```

  - On the execution of these statements, the value of **c** and **d** will be 28 and 65 respectively.

6 Inline Functions

- Example (Eg 2-17)

```
#include<iostream.h>

inline int max(int a,int b)
{
        return a>b?a:b;
}

void main()
{
        int x1=max(3,4);
        int x2=max(7,2);
        int x4=max(x1,x2);
}
```

When compiled, the inline function will be replaced like

```
void main()
{
    int x1=3>4?3:4;
    int x2=7>2?7:2;
    int x4=x1>x2?x1:x2;
}
```

6 Inline Functions

- Inline function
  - To make a function inline is to prefix the keyword **inline** to the function definition.
  - All inline functions must be defined before they are called.

6 Inline Functions

- Inline function
  - We should exercise care before making a function **inline**.
  - The speed benefits of inline functions diminish as the function grows in size.
  - At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost.
  - In such cases, the use of normal functions will be more meaningful.

6 Inline Functions

- Inline function
  - Usually, the functions are made inline when they are small enough to be defined in one or two lines.
    - Example

```
inline double cube (double a)
{
        return (a*a*a);
}
```

6 Inline Functions

- Inline function
  - Remember that
    - The **inline** keyword merely sends a request, not a command, to the compiler.
    - The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

6 Inline Functions

- Some of the situations where inline expansion may not work are
  1. For functions returning values, if a **loop**, a **switch**, or a **goto** exists.
  2. For functions not returning values, if a return statement exists.
  3. If functions contain **static** variables.
  4. If inline functions are recursive.

6 Inline Functions

- Note
  - Inline expansion makes a program run faster because the overhead of a function call and return is eliminated.
  - However, it makes the program to take up more memory because the statement that define the inline function are reproduced at each point where the function is called.
  - So, a trade-off becomes necessary.

6 Inline Functions

- Example (Program 4.1)

7 Default Arguments

- C++ allows us to call a function without specifying all its arguments.
- In such cases, the function assigns a **default value** to the parameter which does not have a matching argument in the function call.
- **Default values** are specified when the function is declared.
- The compiler looks at the prototype to see how many arguments a function uses and alters the program for possible default values.

7 Default Arguments

- An example of a prototype (i.e. function declaration) with **default values**:

```
float amount (float principal, int period, float rate = 0.15);
```

- The **default value** is specified in a manner syntactically similar to a **variable initialization**.

7 Default Arguments

```
float  amount (float  principal, int  period, float  rate = 0.15);
```

- The above prototype declares a **default value** of 0.15 to the argument **rate**.
- A subsequent function call like

```
value =  amount (5000, 7);          //one argument missing
```

- passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for **rate**.
- The call `value =  amount (5000, 5, 0.12);  //no missing argument` passes an explicit value of 0.12 to **rate**.

7 Default Arguments

- A **default argument** is checked for type at the time of **declaration** and evaluated at the time of **call**.
- Note
  - Only the trailing arguments can have default values and therefore we must add defaults from **right to left**.
  - We cannot provide a default value to a particular argument in the middle of an argument list.

7 Default Arguments

- Examples of function declaration with default values

```
int  mul (int  i, int  j = 5,  int  k = 10);        //legal
int  mul (int  i = 5, int  j);                      //illegal
int  mul (int  i = 0, int  j, int  k = 10);         //illegal
int  mul (int  i = 2, int  j = 5, int  k = 10);     //legal
```

7 Default Arguments

- Default arguments are useful in situations where some arguments always have the same value.
  - For instance, bank interest may remain the same for all customers for a particular period of deposit.
- Default arguments also provides a greater flexibility to the programmers.
  - A function can be written with more parameters than are required for its most common application.
  - Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation.

7 Default Arguments

- Example **(program 4.2)**

7 Default Arguments

- Advantages of providing the default arguments are
  1. We can use default arguments to add new parameters to the existing functions.
  2. Default arguments can be used to combine similar functions into one.

8 const Arguments

- In C++, an argument to a function can be declared as **const** as shown below.

```
int strlen (const char *p);
int length (const string &s);
```

- The qualifier **const** tells the compiler that the function should not modify the argument.
  - The compiler will generate an error when this condition is violated.

```
int f(int i1, const int i2)
{
        i1++;                   //valid
        i2++;                   //error
        return i1+i2;
}
```

8 const Arguments

- This type of declaration is significant only when we pass arguments by reference or pointers.

8 const Arguments

- Example **(Eg2_14)**

9 Function Overloading

- Overloading
  - It refers to the use of the same thing for different purposes.
- **Function overloading**
  - In C++ we can use the same function name to create functions that perform a variety of different tasks.
  - This is known as **function polymorphism** in OOP.

9 Function Overloading

- Using the concept of **function overloading**
  - We can design a family of functions with one function name but with different argument lists.
  - The function would perform different operations depending on the argument list in the function call.
  - The correct function to be invoked is determined by checking the **number** and **type** of the arguments but not on the function type.

9 Function Overloading

- Example
  - An overloaded **add()** function handles different types of data

```
//Declarations
int  add(int  a, int  b);                    //prototype 1
int  add(int  a, int  b, int  c);            //prototype 2
double  add(double  x, double  y);           //prototype 3
double  add(int  p, double q);               //prototype 4
double  add(double  p, int  q);              //prototype 5

//Function calls
cout <<  add(5, 10);                 //uses prototype 1
cout <<  add(15, 10.0);              //uses prototype 4
cout <<  add(12.5, 7.5);             //uses prototype 3
cout <<  add(5, 10, 15);             //uses prototype 2
cout <<  add(0.75, 5);               //uses prototype 5
```

9 Function Overloading

- A function call
  - First matches the prototype having the same **number** and **type** of arguments
  - And then calls the appropriate function for execution.
  - The best match must be unique.

9 Function Overloading

- The function selection involves the following steps:
  1. The compiler first tries to find an exact match in which the **types** of actual arguments are the same, and use that function.
  2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as

     char to **int**
     float to **double**

     to find a match.

9 Function Overloading

- The function selection involves the following steps:
  3. When either of the above fails, the compiler
  ① Tries to use the built-in conversion (the implicit assignment conversion) to the actual arguments.
  ② And then uses the function whose match is unique.
  ✓If the conversion is possible to have multiple matches, then the compiler will generate an error message.

9 Function Overloading

- Suppose we use the following two functions

```
long  square (long  n)
double  square (double  x)
```

  - A function call such as `square (10)` will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.

9 Function Overloading

- The function selection involves the following steps:
  4. If all of the steps fail, the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match.
  - ✓ User-defined conversions are often used in handling class objects.

9 Function Overloading

- Example **(Eg2_16)**
- Example **(Program 4.3)**

9 Function Overloading

- Note
  - Overloading of the functions should be done with caution.
  - We should not overload **unrelated functions** and should reserve function overloading for functions that perform **closely related tasks**.
  - Sometimes, the **default arguments** may be used instead of overloading.
  - And this may reduce the number of functions to be defined.
  - Overloaded functions are extensively used for handling **class object**.

10 Friend and Virtual Functions

- **Friend function** and **virtual function** are two new types of functions introduced by C++.
  - They are basically introduced to handle some specific tasks related to **class objects**.

11 Math Library Functions

- The standard C++ supports many math functions that can be used for performing certain commonly used calculations.

Commonly used math library functions

Function	Purposes
ceil(x)	Rounds x to the smallest integer not lesson than x
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x
fabs(x)	Absolute value of x
floor(x)	Rounds x to the largest integer not greater than x
log(x)	Natural logarithm of x (base e)
log10(x)	Logarithm of x (base 10)
pow(x,y)	x raised to power y (x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

11 Math Library Functions

- Note
  - The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.
  - To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

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

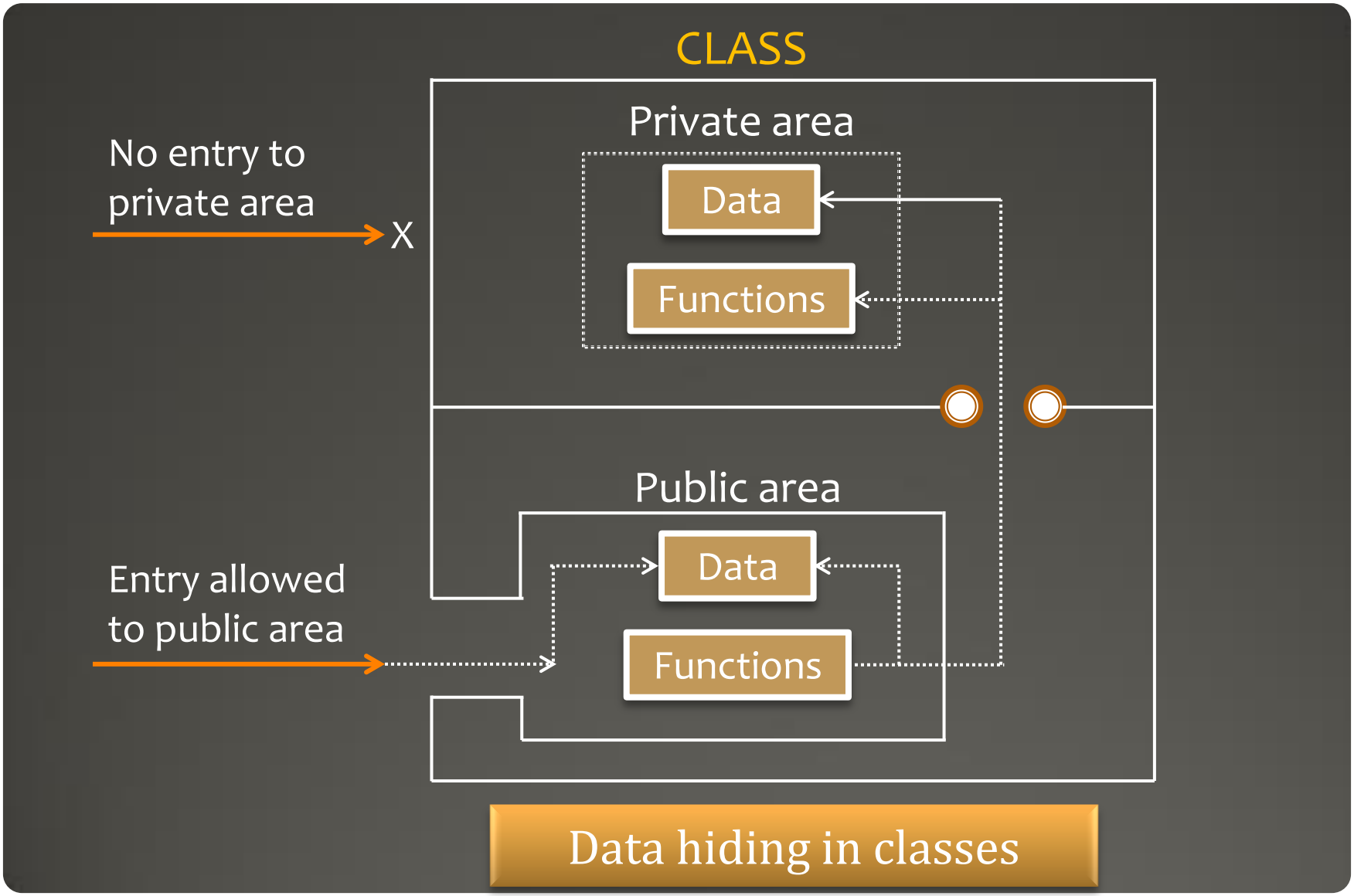
- **Private** & **public**
  - If both the labels are missing, then, be default, all the members are **private**.
  - Such a class is completely hidden from the outside world and does not serve any purpose.

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.


CLASS
Private area
Data
Functions
No entry to
private area
X
Public area
Data
Functions
Entry allowed
to public area
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

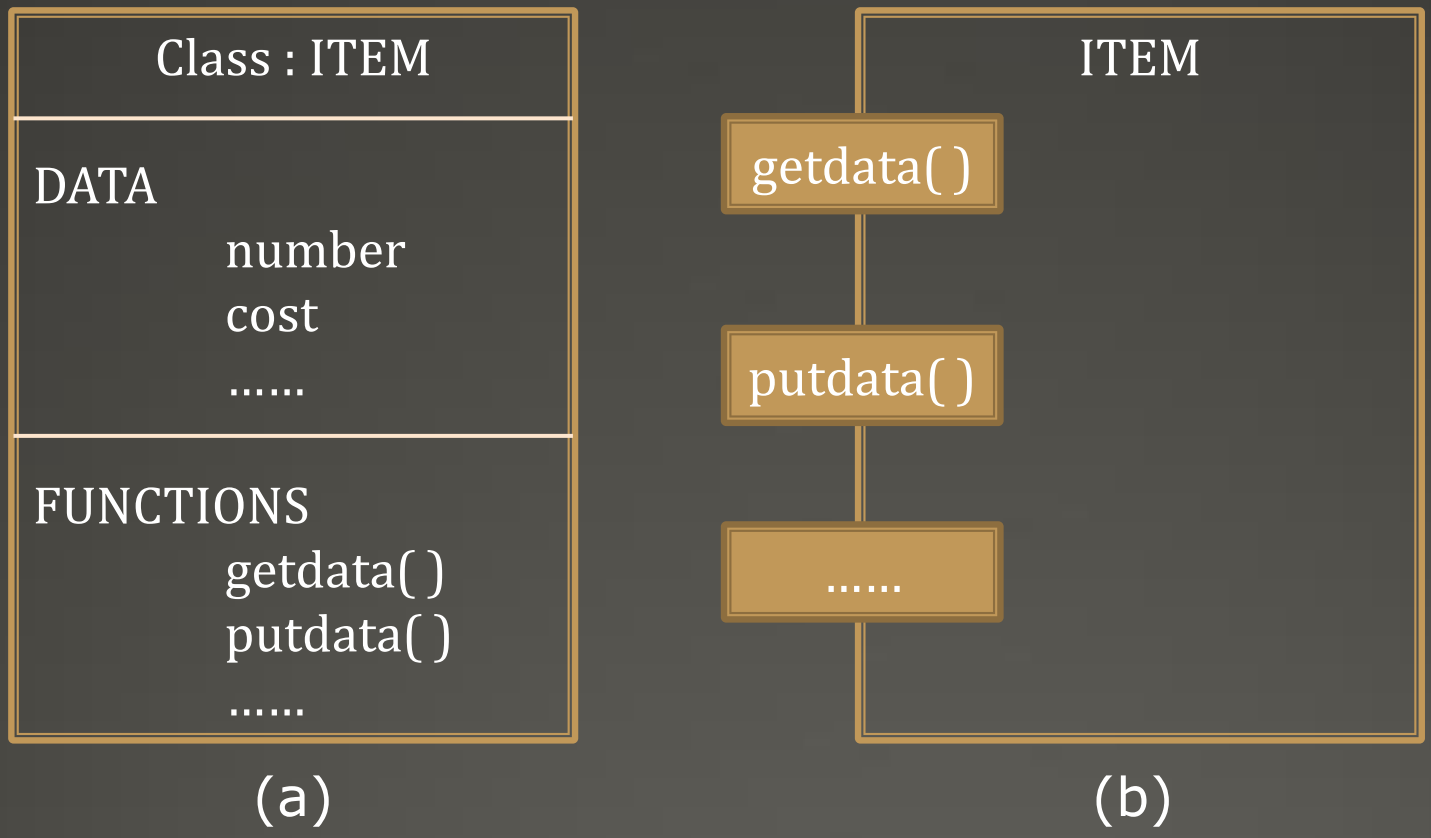
- A Simple Class Example
  - We usually give a class some meaningful name.
  - The name of a class will become a new type identifier that can be used to declare **instances** of that class type.
  - The class item contains two **data members** and two **function members**.
  - The data members are **private**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