P. A. EDUCATIONAL TRUST'S (PAET)

P.A. COLLEGE OF ENGINEERING

MANGALURU -574153 , KARNATAKA (INDIA)

OBJECT ORIENTED CONCEPTS (17CS42)
IV SEMESTER 2018-19

DR ZAHID ANSARI

17CS42: MODULE-I: INTRODUCTION TO OBJECT ORIENTED CONCEPTS:

- ❑ A Review of structures,
- ❑ Procedure–Oriented Programming system,
- ❑ Object Oriented Programming System,
- ❑ Comparison of OO Language with C,
- ❑ Console I/O,
- ❑ variables and reference variables,
- ❑ Function Prototyping,
- ❑ Function Overloading.
- ❑ Class and Objects: Introduction,
- ❑ Member functions and data,
- ❑ Objects and functions,
- ❑ objects and Arrays,
- ❑ Namespaces,
- ❑ Nested classes,
- ❑ Constructors,
- ❑ Destructors.

COURSE OBJECTIVES:

- Learn fundamental features of an object oriented language of C++ and JAVA
- Set up Java JDK environment to create, debug and run simple Java programs.
- Create multi-threaded programs and event handling mechanisms.
- Introduce event driven Graphical User Interface (GUI) programming using applets and swings.

TEXT:

- **TEXT BOOK(S):**

1. Sourav Sahay, Object Oriented Programming with C++ , Oxford University Press,2006 **(Ch 1: 1.1 to 1.9 Ch 2: 2.1 to 2.6 Ch 4: 4.1 to 4.2)**

Chapter 1
Introduction to C++

1.1 REVIEW OF STRUCTURES

- To understand procedural oriented languages, we need to review structure concept.

- Need for Structures – ***value of 1 variable depends on the value of another variable.***

Eg- Date can be programmatically represented in C by 3 different integer variables.

```
int d,m,y;
```

Here: d-date, m-month, y-year

-

- Although 3 variables are not grouped in a code, they actually belong to the same group. The value of 1 may influence the value of other.

- Consider a function ***`nextday()`*** that accepts the addresses of 3 integers that represent a date and changes these values to represent next day.

Prototype of this function:

```
//for calculating the next day
void nextday(int *,int *, int *);
```

Suppose

```
        d=1;
        m=1;
        y=2002;   //1st january 2002
```

If we call `nextday( &d, &m, &y);`

d becomes 2, m=1,y=2002 //2nd January 2002 **(d affected**)

But if

```
d=28;
m=2;
y=1999;//28th Feb 1999
```

and we call the function as

```
nextday( &d, &m, &y);
```

d becomes 1, m will become 3 and y will become 1999.

Again if
```
d=31;
m=12;
y=1999;//31th Dec 1999
```

and we call the function as

```
nextday( &d, &m, &y);
```

d will become 1 , m will become 1 and y becomes 2000.

- A change in 1 variable may change the value of other 2.
- No language construct exist that actually places them in same group.

MEMBERS OF WRONG GROUP MAY BE ACCIDENTALLY SENT TO THE FUNCTION

```
d1=28;   m1=2; y1=1999;  //28thfeb99
d2=19;   m2=3; y2=1999;    //19thmarch99


nextday(&d1,&m1,&y1); //ok
nextday(&d1,&m2,&y2); //incorrect set passed
```

Above listing show problems in passing groups of programmatically independent but logically dependent variables

There is nothing in language itself that prevents the wrong set of variables from being sent to the function.

Suppose ***nextday()*** accepts an array as parameter Then its prototype:

```
void nextday(int *);
```

Let us declare date as an array of 3 integers.

```
int date[3];
date[0]=28;
date[1]=2;
date[2]=1999;    //28th Feb 1999
```

The values of date[0],date[1],date[2] is set to 1,3 and 1999

This method is not convincing. There is no data type of date itself.

The solution to this problem is to create a data type called date itself using structures.

```
struct date d1;
d1.d=28;
d1.m=2; //Need for structures
d1.y=1999;
nextday(&d1);
```

```
struct date{
   int d;
   int m;
   int y;
};
```

- d1.d, d1.m, d1.y will be set correctly to 1,3,1999, since the function takes the address of an entire structure variable as parameter at a time as there is no chance of variables of different groups being sent to the function.
- **Structure is a programming construct in C that allows us to put the variables together.**

- **Library programmers use structures to create new data types.**
- Application programs use these new data types by declaring variables of this data type

```
struct date d1;
```

They call associated functions by passing these variables / addresses to them.

```
d1.d=31;
d1.m=12;
d1.y=2003;
nextday(&d1);
```

They use resultant value of the passed variable further as per requirement.

```
printf(“The next day is: %d/%d/%d\n”,d1.d,d1.m,d1.y);
```

Output

```
The next day is:01/01/2004
```

CREATING A NEW DATA TYPE USING STRUCTURES

Creation of a new data type is a 3 step process.

1. Put structure definition and prototypes of associated functions in a header file.
2. Put the definition of associated functions in a source code and create a library.
3. Provide the header file and library in any media to other programmers who want to use this new data type.

Creating a structure and its associated functions are 2 steps to constitute one complete process

STEP 1:
PUTTING STRUCTURE DEFINITION AND PROTOTYPES OF ASSOCIATED FUNCTIONS IN A HEADER FILE.

```
//date.h contains structure definition &
// prototypes of associated functions
struct date
{
 int d, m, y;
}
void nextday(struct date *);
void getsysdate(struct date *);
```

STEP 2:
PUT DEFINITION AND OTHER PROTOTYPES IN A SOURCE CODE AND CREATE A LIBRARY

```
#include "date.h"
void nextday(struct date *p)
{//calculate date represented by *p and set it to
  *p}
void getsysdate(struct date *p){ // determine
  //current system date & set it to *p}
  //definitions of other useful & other relevant
  //functions to work upon variables of date
  structure
```

USING STRUCTURES IN APPLICATION PROGRAMS IS A 3 STEP PROCEDURE

1. Include header file provided by programmer in the source code.
2. Declare variables of new data type in the source code.
3. Embed calls to the associated functions by passing these variables in the source code.
4. Compile the Source code to get the object file.
5. Link the Object file with the library provided by the library programmer to get the executable or another library.

STEP1
INCLUDE THE HEADER FILE PROVIDED BY THE PROGRAMMER IN THE SOURCE CODE

```
//beginning of dateuser.c
#include “date.h”
void main()
{
…
…
}//end of dateuser.c
```

STEP 2
DECLARE VARIABLES OF NEW DATA TYPE IN THE SOURCE CODE.

```
//beginning of dateuser.c
#include “date.h”
void main()
{  struct date d;
…
…
}//end of dateuser.c
```

STEP 3
EMBED CALLS TO ASSOCIATED FUNCTIONS BY PASSING THESE VARIABLES IN SOURCE CODE

```
//beginning of dateuser.c
#include “date.h”
void main()
{
struct date d;
d.d=28;  d.m=2; d.y=1999;
nextday(&d);
…
}  //end of dateuser.c
```

1.2 PROCEDURE ORIENTED PROGRAMMING SYSTEM

Procedure Oriented System has the following features

- Program codes are **divided into** smaller programs known as **functions**
- **Focus** is on **functions**.
- Functions can **share global data**
- Data can move freely around the system from one function to other function. Data is passed from one function to another to be read from or written into.
- Functions are associated with the data but they are not a part of it. Instead they receive variables / their addresses and work upon them.
- Employs **top-down approach** in program design.

Drawback/Disadvantage

1. **Data is not secure** and can be manipulated by any function/procedure.
2. **Associated functions** that were designed by library programmer **don’t have exclusive rights** to work upon the **data**.
3. Application program might modify the data by some code inadvertently written in application program itself

.

Consider an application of around 25,000 lines in which the variables of structure is used quite extensively

1. Testing may find that date being represented by one of these variables has become 29th Feb 1999.
2. This faulty piece of code can be anywhere in the program.
3. Hence Debugging will involve a visual inspection of the entire code & will not be limited to associated functions only.
4. While distributing his/her application, application programmer cant be sure that program would run successfully.
5. Every new piece of code accessing structure variable will have to be inspected/tested again to ensure that it doesn't corrupt the structure.
6. Compilers that implement procedure oriented programming systems don't prevent unauthorized functions from accessing the structure variables.

7. To ensure a successful compilation of code, application programmer is forced to remove those statements that access data members of structure variables.

8. Lack of data security of procedure oriented programs has led to Object Oriented Programming Systems

1.3 OBJECT ORIENTED PROGRAMMING SYSTEMS

Model real-world objects (RWO)

- RWO has internal parts & interfaces that enable us to operate them.
- Eg: LCD is RWO-has a fan and a lamp.

  There are 2 switches 1 to operate fan & other to operate lamp.

  Switch operation has rules.

  - If lamp is switched on, fan is automatically switched on, else LCD will be damaged.
  - Lamp is also switched off if fan is switched off and switches are linked with each other.

Common Characteristic of RWO

- If a perfect interface is required to work on an object , it will have exclusive rights to do so.
- Coming to C++, observed behaviour of LCD projector resembles the desired behaviour of the date's structure variables.

OO PROGRAMMING SYSTEMS

- **Emphasis is on data** rather than procedures or functions.
- Programs are **divided into** what are known as **objects.**
- Data structures are designed such that they **characterize the objects**.
- **Functions** that operate on the **data** are **tied together** in the data structure
- **Data is hidden** and cannot be accessed by external functions.
- Objects may **communicate** with each other through functions.
- **New data** and functions **can be easily added** whenever necessary.
- Follows **bottom-up approach** in program design.
- Compilers implementing OOPs **enable data security** enforcing by throwing compile-time errors against the pieces of code.
- RWO ensure a **guaranteed initialization of objects**

OOP FEATURES/CHARACTERISTICS

1. Class
2. Object
3. Encapsulation
4. Data Hiding and Abstraction
5. Inheritance
6. Polymorphism
7. Message Passing
8. Dynamic Binding etc.

- **CLASS:** Class is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Eg: grapes bannans and orange are the member of class fruit. Example:

  Fruit orange; // Here object orange is an instance of class fruit.

- **OBJECT:** Object is a collection of attributes/entities. Objects take up space in the memory. Objects are instances of classes. When a program is executed , the objects interact by sending messages to one another. Each object contain data and code to manipulate the data. Objects can interact without having know details of each others data or code.

- **ENCAPSULATION :**
- Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**. Data encapsulation is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those function which are present in the class can access the data.
- **Data Hiding and Abstraction**
- The insulation of the data from direct access by the program is called **data hiding**. Hiding the complexity of program is called **Abstraction** and only essential features are represented. In short we can say that internal working is hidden.

INHERITANCE

- **INHERITANCE:** it is the process by which object of one class acquire the properties or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without Modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes.

- <u>Example:</u> Sparrow is a part of the class flying bird which is again a part of the class bird.

In Inheritance, both data and functions may be inherited

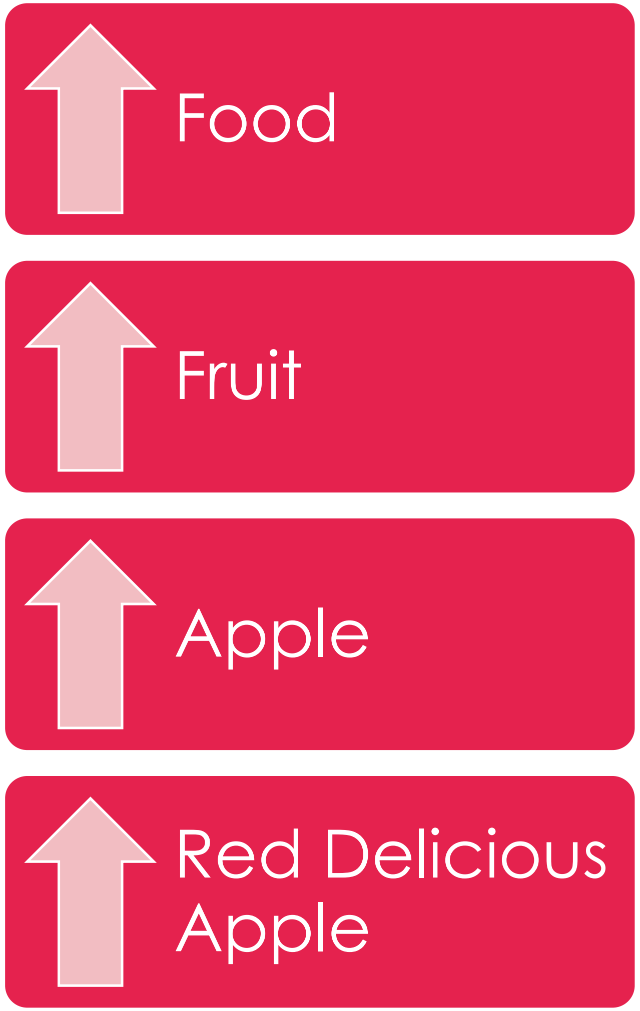
- Parent class can be given the ***general*** characteristics, while its child may be given more ***specific*** characteristics.
- Inheritance allows code ***reusability*** by keeping code in a common place – the base structure.
- Inheritance allows code ***extensibility*** by allowing creation of new structures that are suited to our requirements compared to existing structures.

Inheritance – a process by which 1 object can acquire the properties of another object. This is important as it supports classification.

- Most knowledge is made of hierarchical Classification.

  Eg-Red delicious apple is part of apple classification which in turn is a part of fruit class, which is under the larger class food.

- Inheritance mechanism makes it possible for one object to be a specific instance of a more general class.



Food

Fruit

Apple

Red Delicious Apple

POLYMORPHISM

- **POLYMORPHISM:** A greek term means ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.
  - **Function overloading & Operator Overloading**
  - **Static & Dynamic Polymorphism**

- **MESSAGE PASSING:** The process by which one object can interact with other object is called message passing.

- **DYNAMIC BINDING:** Refers to linking of function call with function definition is called binding and when it is take place at run time called dynamic binding.

C	C++
1. C compiler cannot execute C++ programs 2. In C, u may / may not include function prototypes	1. C++ compiler can execute C programs 2. In C++, you must Include function prototypes
3. C doesn't allow for default arguments	3. C++ lets you to specify default arguments in function prototype
4. Declaration of the variables must be at the beginning	4. Declaration of the variables can be anywhere before using
5. If a C program uses a Local variable that has Same name as global variable, then C uses the value of a local variable. 6. Function overloading is not there. 7. Function inside the structure is not allowed 8. Object initialization doesn't exist 9. Data hiding, data abstraction and data encapsulation feature doesn't exist	5. In C++, you can instruct program to use value of global variable with scoperesolution Eg- cout << "I am global var :" << ::l; 6. Function overloading exists 7. Function inside the structure is allowed 8. Object initialization (constructor) exist 9. Data hiding, data abstraction and data encapsulation exists in C++

CONSOLE INPUT-OUTPUT IN C++

1. **Console output**

i. cout << constant/variable

ii. cout<<endl;

iii. cout<<“\n”;\\newline

2. **Console input**

i. cin>> variable

VARIABLES IN C++

- Can be declared anywhere in the C++ program before using them.

REFERENCE VARIABLE

- Used as **aliases** for other variables within a function.
  - All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable.
  - An alias is simply another name for the original variable.
  - Must be initialized at the time of declaration.

- Example

```
int count = 1;
int & iRef = count;
iRef++;
```

  - Increments count through alias *iRef*

REFERENCE VARIABLE

```
#include <iostream.h>

void main()
{
   int x=3;
   int &y=x;
   cout <<"x="<<x endl<<"y="<<y<<endl;
   y=7;
   cout <<"x="<<x <<endl<<"y="<<y<<endl;
 }
```

Creating a reference as an alias to another variable in the function

Assign **`7`** to **`x`** through alias **`y`**

```
OP:
x=3
y=3
x=7
Y=7
```

REFERENCE VARIABLE

- **The value of a reference variable can be read in the same way as the value of an ordinary variable is read. Consider the following:**

```
#include <iostream.h>
void main()
{
   int x, y;
   int x=100;
   int & iRef=x;
   y=iRef;
   cout <<y<<endl;
   y++; // x and iRef unchanged
   cout <<x <<endl<<iRef<<endl<<y<<endl;
 }
```

```
100
100
100
101
```

CALL BY REFERENCE

- Reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.
- An illustrative example follows:

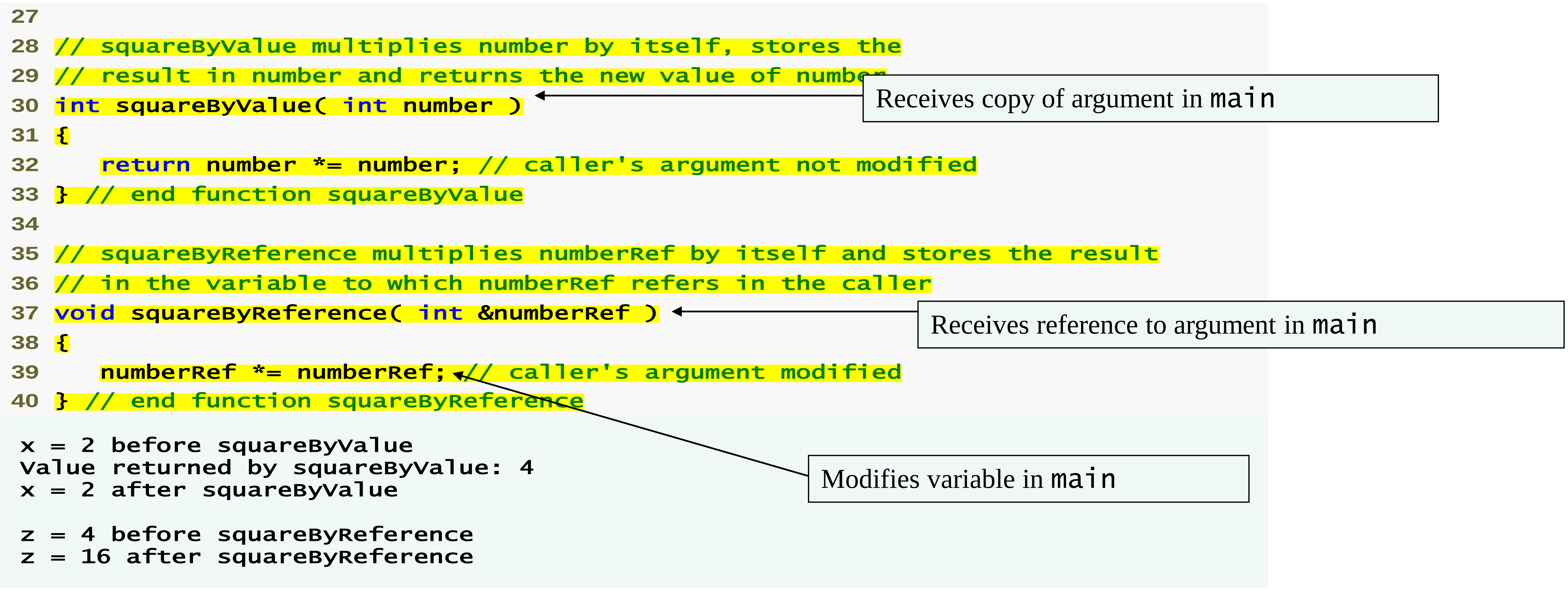
```cpp
// Fig. 18.5: fig18_05.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
   return 0; // indicates successful termination
} // end main
```

Function illustrating pass-by-value

Function illustrating pass-by-reference

Variable is simply mentioned by name in both function calls

Call by Reference and Call by Value in C++



```
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in the caller
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`

RETURN BY REFERENCE

- Functions can return by reference also.
- An illustrative example follows:

```
#include <iostream.h>
int & large (int &, int &);
void main()
{
   int a,b;
   a=5; b=10;
    int & r = large (a,b);
    r=-1;
      cout <<a <<endl<<b<<endl;
 }
int & large ( int & x, int & y)
{
  if (x>y)
        return x;
 else
        return y;
}
O/P:
5
-1
```

RETURN BY REFERENCE

- In the example program , variable x and a refer to the same location , while y and b refer to the same location.
- From the large () function , a reference to y, that is reference to b is returned and stored in reference variable r.
- The function large() does not return the value y because the return type is int & and not int.
- Thus the address of r becomes equals to the address of b. So, any change in value of r also changes the value of b

RETURN BY REFERENCE

- The previous program can be shortened as:

```
#include <iostream.h>
int & large (int &, int &);
void main()
{
   int a,b;
   a=5; b=10;
   large (a,b)=-1;
   cout <<a <<endl<<b<<endl;
 }

Output
5
-1
```

- A function that returns by reference, primarily returns the address of the returned variable

ASSIGNMENT (=) OPERATOR WITH VARIOUS CASES

- If the compiler finds a non constant variable on the left side of the ‘=‘ operator , it does the following actions:
  - Determine the address of the variable
  - Transfer the control to the byte that has that address and
  - Write the value on the right of the ‘=‘ operator into the block that begins with the byte found above.

- If a function call is found on the left side of the ‘=‘ operator , it does the following actions:
  - Transfer the control to the byte whose address is returned by the function
  - Write the value on the right of the ‘=‘ operator into the block that begins with the byte found above.

- If the compiler finds a variable on the right side of the ‘=‘ operator , it does the following actions:
  - Determine the address of the variable
  - Transfer the control to the byte that has that address
  - Read the value from the block that begins with the byte found above
  - Push the read value in to the stack

- If the call is found on right side of the ‘=‘ operator , compiles does the following actions:
  - Transfer the control to the byte whose address is returned by the function
  - Read the value from the block that begins with the byte found above
  - Push the read value in to the stack

- A constant cannot be placed on the left of the assignment operator. This is because constants do not have a valid address.
- A call to a function that returns by reference can be placed on the left side of an assignment operator.

RETURN BY REFERENCE

- We must avoid returning a reference to a local variable , because it can lead to run-time errors

Example:

```
#include <iostream.h>
int & abc ();
void main()
{
   abc() = -1;
 }
int & abc ()
{
   int x;
   return x;
 }
```

- The problem here is x will go out of scope .
- Thus the statement abd()=-1 will write -1 in an unallocated block of memory.

FUNCTION PROTOTYPE, FUNCTION OVERLOADING

FUNCTION PROTOTYPING

- Function Prototyping is necessary in C++.
- C++ strongly supports function prototypes.
- Prototype describes the function's interface to the compiler
- Tells the compiler the return type of function, number , type and sequence of its formal arguments
- ***General syntax:***

```
    return-type function-name (argument_list);
    Eg: void interest (float, int, float);
```

- Since function prototype is also a statement , a semicolon must follow it.
- Providing names to the formal arguments in function prototype is optional.

With prototyping , compiler ensures following

1. The return value of a function is handled correctly.
2. Correct number and type of arguments are passed to a function.

What happens in the absence of prototype?

1. Suppose, compiler assumes that the type of the returned value is an integer. However, the called function may return a value of an incompatible type(say **struct** type) .
2. Now , suppose an **int** variable is equated to a function call where the function call proceeds the function definition . In this situation, the compiler will report an error against the function definition and not the function call
3. However, if the function definition is in different file to be compiled separately, then no compile time errors will arise. Instead , wrong results will arise during run-time (weird results)

- Since C++ compiler require function prototyping, it will report error against function call because no function prototype is provided to resolve the function call.

- Compiler may still give an error, if function call doesn’t match the prototype.

- **Hence prototyping guarantees protection from errors arising out of incorrect function calls.**

- **Function Prototyping produces automatic type of conversion wherever appropriate.**

- For example, if the compiler expects an **int** type value, but a type of **double** value is wrongly passed. During run time, the value in first 4 bytes of the passed eight bytes is extracted. This is obviously undesirable.
- However, C++ compiler converts **double** type to **`int`** type automatically , this this because of the function prototype before the function call.
- Also, the automatic type conversion takes place only when it makes sense. For eg, struct to int is not possible.

DEFAULT ARGUMENTS

- Default values are given in the function prototype declaration.
- Default arguments must be specified only in function prototypes. They should not be specified in function definitions
- Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters.
- Default values are always **assigned from right to left.**

➢For example :

```
➢ void interest (float p, int t, float r=0.25); //legal
➢void interest (float p,  float r=0.25, int t); //illegal
➢void interest (float p, int t=10, float r=0.25); //legal
```

```
void sum(int, int, int=6,int=10);
void main( ) {
    int a, b, c, d;
    cout<<“enter any two numbers.\n”;
    cin >>a >>b;
    sum (a,b);
}
void sum (int a1, int a2, int a3, int a4) {
    int temp;
    temp=a1 + a2 + a3 +a4;
    cout << “sum=“ <<  temp;
}
```

Output:

```
    enter any two numbers:
       11   21
       sum=48
```

If input is given for c and d, then default values will not be used.

DEFAULT ARGUMENTS

```
 void DisplayStars(int = 10, int =1);
 void main( )
 { DisplayStars( );      cout << endl;
    DisplayStars(5);     cout << endl;
    DisplayStars(7,3);
 }

void DisplayStars(int Cols, int Rows) {
 for (int Down = 0; Down < Rows; Down++) {
       for (int Across=0; Across < Cols;Across++)
             cout << '*';
         cout << endl;
}   }
```

DEFAULT ARGUMENTS (CONTD…)

- Care is taken in specifying default arguments in function overloading; otherwise and ambiguity might occur.

  For eg:

-
  ```
  int add (int, int, int =0);
  int add (int, int)
  ```

  Can confuse the compiler

FUNCTION OVERLOADING

- Overloading refers to the use of same thing for different purposes.
- We can use the same function name to create functions that performs variety of different tasks, called function overloading (polymorphism).
- For example a function add() can be overloaded as:

```
int add(int a, int b);

int add(int a, int b, int c);

float add(float a, float b);
```

FUNCTION OVERLOADING (CONTD…)

- A function call **first matches** the prototype having the same **number and type of argument** and then calls the appropriate function for execution.
- The function selection involves the following steps:

1. The compiler first tries to find the exact match in which the type of actual arguments are same, and use that function
2. If the exact match is not found , then the compiler uses the **integral promotions** to the actual parameters such as:

   char to **int**
   int to **long** or **float**
   float to **double** to find a match

FUNCTION OVERLOADING (CONTD…)

3. If the conversion is possible to have multiple matches then the compiler will generate an error message (ambiguous situation).

Eg:

```
    long square(long n)
    float square(float n)
```

A call such as: square(10)

COMPUTE AREA OF SQUARE, RECTANGLE AND TRIANGLE

```
 void main() {
   int area(int);
   int area(int, int);
   float area(float, float);
    cout<<area(10)<<"\n";
    cout<<area(10,5)<<"\n";
    cout<<area(10.5, 5.5)<<"\n";
}
int area( int s) {  return s*s; }
int area( int a, int b) {  return a*b;}
float area(float a, float b) { return (0.5*a*b); }
```

COMPUTE volume of cube, cylinder and cuboid

```
void  main( ) {
   cout << volume(10) << “\n”;
   cout << volume(2.5, 8) << “\n”;
   cout << volume(100L, 75, 15) << “\n”;
}
// Function definitions
int volume(int s) {  return(s*s*s);  }
double volume(double r, int h) {  return(3.14*r*r*h); }
long volume(long l, int b, int h) {  return(l*b*h); }
```

Output:

```
1000
157.26
112500
```

Chapter 2
Classes and objects

INTRODUCTION TO CLASSES AND OBJECTS

- Classes are to C++ while structure are to C. Both provide the library programmer a means to create new data types.

- In C structure there is no guarantee that the client programs will use only the functions which manipulate the members of variables of the structure.

- Note that in C, there is ***no facility to bind the data and the code*** that can have the exclusive rights to manipulate the data. This may lead to run-time bugs.

- The C compiler does not provide the library programmer with the facilities such as: **data encapsulation, data hiding and data abstraction**.

INTRODUCTION TO CLASSES AND OBJECTS

- C++ compiler provides a solution to the problem by redefining the structure , which allow member functions also.

```
struct Rectangle
{
          int width;
          int length;
          void setwidth(int w) (width=w;}
          void setlength(int l) (length=l;}
          int area() { return length*width; }
};
```

INTRODUCTION TO CLASSES AND OBJECTS

```
void main( ) {
     Rectangle r1,r2;
     r1.setwidth(3);
     r1.setlength(5);
     r2.setwidth(4);
     r2.setlength(6);
     cout<<“Area of r1:”<<r1.area();
     cout<<“Area of r2:”<<r2.area();
}
```

INTRODUCTION TO CLASSES AND OBJECTS

- It is possible to define functions within the scope of the structure definition. Because of this, not only the member data of the structure can be accessed through the variables of the structures but also the member functions can be invoked.

- For example , r1.setwidth(2) and r1.setlength(5) assigns r1.width and r1.length to 2 and 5 respectively.

PRIVATE AND PUBLIC MEMBERS

- Advantage of having member function in structure is, we can put data and functions that work on the data together. But the problems in ***code debugging*** can still arise.
- Specifying the member functions as public but member data as private gives the advantage.

```
struct Rectangle {
private: int width;
      int length;
public:
      void setwidth (int w) {width=w;}
      void setlength (int l) {length=l;}
      int area() {return length*width;}
};
```

PRIVATE AND PUBLIC MEMBERS

- Upon declaring the **member data under private part** and **member functions under public part** tells the compiler that width and length are private data members of the variables of the structure Rectangle and member functions are public.
- The values of data members width and length can be accessed/modified only through the member functions of the structure and not by non member functions.

```
void main( ) {
    Rectangle r1,r2;
    r1.setwidth(3);
    r1.setlength(5);
    r1.width++; //error , non-member function accessing private data
    r2.setwidth(4);
    r2.setlength(6);
    cout<<“Area of r1:”<<r1.area();
    cout<<“Area of r2:”<<r2.area();
 }
```

PRIVATE AND PUBLIC MEMBERS

- Thus the values of width and length can only be accessed/modified by member functions of the structure and not by non member function. Compiler gives an error message, if any violation made to this restriction.

- The **private** and **public** keywords are known as access modifiers or **access specifiers**.

- **C ++** introduces a new keyword **class** as a substitute to the keyword **struct.**

- In a structure , members are **public by default** . On the other hand class members are **private by default.**

PRIVATE AND PUBLIC MEMBERS

- The structure Rectangle can be redefined by using class keyword. Example:

```
class Rectangle {
      int width;// private by default
      int length; // private by default
   public:
         void setwidth(int w) {width=w;}
         void setlength(int l) {length=l;}
         int area() { return length*width; }
};
```

- The **struct** keyword is has been retained to maintain **backward compatibility with C**.

OBJECTS

- Variables of the class are known as objects.
- The memory requirement of object of class and variable of structure are same, if both have same data members.
- Introducing member functions does not influence the size of objects.
- Moreover, making data members private or public does not influence the size of objects.

OBJECTS

Example:

```
struct A {
   char c;
   int i;
};
class B {
   char c;
   int i;
};
void main(){
   cout<<sizeof(A)<<endl<<sizeof(B);
}
```

SCOPE RESOLUTION OPERATOR

- The scope resolution operator (::) makes it possible for the library programmer to define the member functions outside their respective classes.

- The scope resolution operator signifies the class to which the member functions belong.

- The class name is specified on the left-hand side of the scope resolution operator. The name of the function being defined is on the right-hand side.

SCOPE RESOLUTION OPERATOR

Example:

```
class Rectangle {
            int width;// private by default
            int length; // private by default
      public:
            void setwidth (int ); //prototype
            void setlength(int ) ; //prototype
            int area(); //prototype
};
void Rectangle::setwidth (int w) { width=w;} // definition
void Rectangle::setlength (int l) {  length=l;} // definition
int Rectangle::area() { return length*width; } //definition
```

CREATING LIBRARIES USING THE SCOPE RESOLUTION OPERATOR

Creating a new data type in C++ using classes is a three-step process that is executed by the library programmer.

Step 1 Place the **class definition** in a **header file**.

Step 2 Place the **definitions of the member functions** in a C++ **source file** (the library source code). A file that contains definitions of the member functions of a class is known as the **implementation file** of that class. Compile this implementation file and put in a library.

Step 3 Provide **the header file and the library**, in whatever media, to other programmers who want to use this new data type.

USING CLASSES IN APPLICATION PROGRAMS

STEPS FOLLOWED BY PROGRAMMERS FOR USING THE NEW DATA TYPE

Step 1 Include the header file provided by the library programmer in their source code.

Example:

```
    #include <Rectangle.h>
    void main () {
      …………..
      …………..
    }
```

Step 2 Declare variables of the new data type in their source code.

Example:

```
    #include <Rectangle.h>
    void main () {
      Rectangle r1,r2;
      …………..
    }
```

Step 3 Embed calls to the associated functions by passing these variables in their source code.

Example:

```
#include <Rectangle.h>
void main () {

    Rectangle r1,r2;
    r1.setwidth(3);
    r1.setlength(5);
    r2.setwidth(4);
    r2.setlength(6);
    cout<<“Area of r1:”<<r1.area();
    cout<<“Area of r2:”<<r2.area();
}
```

Step 4 Compile the source code to get the object file.

Step 5 Link the object file with the library provided by the library programmer to get the executable or another library.

THE *THIS* POINTER

- The C++ compiler creates and calls member functions of class objects by using a unique pointer known as the ***this*** pointer.
- It is always a **constant** pointer.
- The ***this*** pointer always points at the object with respect to which the function was called.

WORKING OF THIS POINTER

- Once the compiler is sure that no attempt is made to access the private members of an object by nonmember functions, it converts C++ code into an ordinary C language code as follows.

1. IT CONVERTS CLASS INTO STRUCTURE WITH ONLY DATA MEMBERS AS FOLLOWS

Before

```
class Rectangle {
          int width;// private by default
          int length; // private by default
       public:
          void setwidth (int ); //prototype
          void setlength(int ) ; //prototype
          int area(); //prototype
};
```

After

```
struct Rectangle {
          int width;
          int length;
};
```

2. IT PUTS A DECLARATION OF THE CONSTANT THIS POINTER AS A LEADING FORMAL ARGUMENT IN THE PROTOTYPES OF ALL MEMBER FUNCTIONS AS FOLLOWS

```
Before –
  void setwidth (int );
After -
  void setwidth (Rectangle * const , int);
Before –
  void setlength(int )
After –
  void setlength (Rectangle * const , int );
Before –
  int area();
After –
  int area(Rectangle * const );
```

3. It puts the definition of this pointer as a leading formal argument in the definitions of all member functions.

It also modifies all statements to access object members by accessing through the this pointer using the pointer-to-member access operator (->).

Before –

```
void Rectangle :: setlength ( int x) {
     length = x;
}
```

After –

```
void setlength( Rectangle* const  this, int x) {
     this -> length = x;
}
```

Before -

```
void Rectangle :: setwidth( int x){ width = x; }
```

After -

```
void setwidth( Rectangle* const this , int x) {
     this -> width = x;
}
```

Before -

```
int Rectangle:: area() { return length * width; }
```

After -

```
int getFeet( Rectangle * const this ) {
     return (this->length * this->width);
}
```

4. PASSES THE ADDRESS OF INVOKING OBJECT AS LEADING PARAMETER TO EACH CALL TO THE MEMBER FUNCTION AS FOLLOWS

Before –

```
r1.setwidth(3);
```

After -

```
setwidth(&r1, 3);
```

Before –

```
r1.setlength(5);
```

After –

```
setlength(&r1 , 5);
```

- Its evident that ‘this’ pointer should continue to point at same object , the object with respect to which the member Function has been called throughout the lifetime.

- **Hence the compiler creates it as a constant pointer.**

```
class Distance  { //Distance.h
      int feet;
      float inches;
public:
      void setFeet(int);     //only member function
      int getFeet();         //prototypes are given
      void setInches();     //in the class definition.
      float getInches();
      Distance add( Distance);
 };
```

```
Distance add( Distance dd) {
   Distance temp;
   temp.feet = feet+dd.feet;
   temp.inches=inches+ dd.inches;
   return temp;
}
// described conversion for this add() using this ptr
Distance add(Distance * const this , Distance dd) {
 Distance temp;
 temp.feet = this->feet+dd.feet;
 temp.inches=this->inches+ dd.inches;
 return temp;
}
```

A statement d3=d1.add(d2) is invoked as d3=add(&d1,d2);

DATA ABSTRACTION

- **Data abstraction is a virtue by which an object hides its internal operations from the rest of the program.**
- Data abstraction makes it unnecessary for the client programs to know how the data is internally arranged in the object.
- This obviates the need for the client programs to write precautionary code upon creating and while using objects. And **Data abstraction is effective due to data hiding only**.
- Perfect definitions of the member functions are guaranteed to achieve their objective because of data hiding. This is the essence of the object-oriented programming system.
- Real-world objects have not only working parts but also an exclusive interface to these inner-working parts. A perfect interface is guaranteed to work because of its exclusive rights.

DATA ABSTRACTION

- Consider a Distance class. The library programmer, who has designed the 'Distance' class, wants to ensure that the 'inches' portion of an object of the class should never exceed 12. If a value larger than 12 is specified by an application programmer while calling the '`Distance::setInches()`' function, the logic incorporated within the definition of the function should automatically increment the value of 'feet' and decrement the value of 'inches' by suitable amounts.

- **Example:** A modified version of ‘Distance::setInches’ function is as follows:

```
void Distance::setInches(int i) {
     inches=i;
     if (inches>=12) {
          feet=feet+inches/12;
          inches=inches%12;
     }
 }
```

DATA ABSTRACTION

- Now, we notice that an application programmer need not send inch values always less than 12 while calling the function ***'Distance::SetInches()'*** . By default the logic of the function does necessary adjustments.

- Similarly, the definition of the ***'Distance::add()'*** function should also be modified as follows by the library programmer. i.e., either an invoking object's or argument object's 'inches' portion can be greater than 12.

```
Distance Distance::add(Distance d) {
         Distance temp;
         temp.feet = feet+d.feet;
         temp.Setinches(inches+ d.inches);
         return temp;
}
```

- Note that the data abstraction is effective due to data hiding only. Perfect definitions of member functions are guaranteed to achieve their objective because of data hiding

ARROW OPERATOR

- Member functions can be called with respect to an object through a pointer pointing at the object. The arrow operator (**->**) does this.
- The definition of the arrow (**->**) operator has been extended in C++. It takes not only data members on its right as in C, but also member functions as its right-hand side operand.

ARROW OPERATOR

Example:

```
#include <iostream.h>
#include <Distance.h>
void main () {
      Distance d, *dptr;
      dptr= &d;
      dptr->setFeet(2);
      dptr->setInches(10);
      cout<<dptr->getFeet()<<“\t”<<dptr->getInches();
}
```

If the operand on its right is a data member, then the arrow operator behaves just as it does in C language. However, if it is a member function of a class, then the compiler simply passes the value of the pointer as an implicit leading parameter to the function call. For example:

dptr->setFeet(2) after conversion become: setFeet(dptr,2) // ‘dptr’ value is copied to ‘this’ pointer.

CALLING ONE MEMBER FUNCTION FROM ANOTHER

- One member function can be called from another. **Example:**

```
class A {
  int a;
public:
   void seta(int);
   void setaindirect(int);
};
void A::seta(int p) { a=p;}
void A::setaindirect(int q) { seta(q);}
void main(){
   A a1;
   a1.seta(10);
   a1.setaindirect(20);
}
```

EXPLICIT ADDRESS MANIPULATION

- An application programmer can manipulate the member data of any object by explicit address manipulation.

```
#include“Distance.h”
#include<iostream.h>
void main() {
   Distance d1;
   d1.setFeet(256);
   d1.setInches(2.2);
   char * p=(char *)&d1; //explicit address manipulation
   *p=1; //undesirable but unpreventable
   cout<<d1.getFeet()<<“ ”<<d1.getInches()<<endl;
}
```

Output

```
257 2.2
```

OVERLOADED MEMBER FUNCTIONS

- Member functions can be overloaded just like non-member functions.
- Function overloading also enables us to have two functions of the same name and same signature in **two different classes**.
- Without the facility of function overloading, choice of names for member functions would become more and more restricted.
- Function overloading enables **function overriding** that, in turn, enables **dynamic polymorphism**

FUNCTION OVERRIDING

In C++, overriding is a concept used in inheritance which involves a base class implementation of a method.

Then in a subclass, you would make another implementation of the method.

This is overriding. Here is a simple example:

```
class Base {
 public:
 virtual void DoSomething() {x = x + 5;}
 private:
 int x;
};
class Derived : public Base {
 public:
 virtual void DoSomething() {
     y = y + 5;
     Base::DoSomething();
 }
 private:
 int y;
};
```

FUNCTION OVERRIDING CONTINUED …

- Here you can see that the derived class overrides the base class method **`DoSomething`** to have its own implementation where it adds to its variable, and then invokes the parent version of the function by calling **`Base::DoSomething()`** so that the x variable gets incremented as well. The virtual keyword is used to that the class variable can figure out which version of the method to call at runtime.

OVERLOADING

Overloading is when you make multiple versions of a function. The compiler figures out which function to call by either

1) The different parameters the function takes or
2) the return type of the function. If you use the same function declaration, then you will get a compiler error because it will not know which function to use. Example.

```
class SomeClass{
public:
      void SomeFunction(int &x) { x *= x; }
      int SomeFunction(int x) { return x * x; }
};
// In main()
SomeClass s;
int x = 5;
x = SomeFunction(x); // Second method is called
```

DEFAULT VALUES FOR FORMAL ARGUMENTS OF MEMBER FUNCTIONS

- We already know that default values can be assigned to arguments of non-member functions. Default values can be specified for formal arguments of member functions also.
- Giving default values to arguments of overloaded member functions can lead to **ambiguity errors.**
- If default values are specified for more than one formal argument, they must be specified from the right to the left.
- Default values must be specified in the function prototypes and not in function definitions.
- Default values can be specified for a formal argument of any type.

INLINE MEMBER FUNCTIONS

- Member functions are made inline by either of the following two methods:

1. By defining the function within the class itself .
2. By only prototyping and not defining the function within the class.
    - The function is defined outside the class by using the scope resolution operator.
    - The definition is prefixed by the inline keyword.

- As in non-member functions, the definition of the inline function must appear before it is called.
- The function should be defined in the same header file in which its class is defined.

Member functions are made inline by either of the following two methods:

1. By defining the function within the class itself as given below

```
class A {
public:
        void show() {/* definition of show inside the class */} //
};
```

2. By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the inline keyword. The function should be defined in the same header file in which its class is defined. Example:

```
class A {
public:
        void show();
};
inline void A::show() { //definition in header file itself
//definition of A::show() function
}
```

CONSTANT MEMBER FUNCTIONS

- Member functions are specified as constants by suffixing the prototype and the function definition header with the **const** keyword.
- For constant member functions, the memory occupied by the invoking object is a read-only memory as the this pointer becomes ‘a constant pointer to a constant’ instead of only ‘a constant pointer’.
- Only constant member functions can be called with respect to constant objects. (here ‘this’ pointer becomes the ‘constant pointer to a constant’., i.e. ,for e.g., const Distance *const)
- Non-constant member functions cannot be called with respect to constant objects.
- Constant as well as non-constant functions can be called with respect to non-constant objects.

CONSTANT MEMBER FUNCTIONS

- It is possible to overload a function in such a way to have a const and non-const version of the same function:

```
class Something {
  public:
        int value;
        const int& GetValue()  const { return value; }
        int& GetValue() { return value; }
 };
```

- The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects. Example:

```
Something cSomething;
cSomething.GetValue(); // calls non-const GetValue();
const Something cSomething2;
cSomething2.GetValue(); // calls const GetValue();
```

CONSTANT MEMBER FUNCTIONS CONTINUED

- **const (Constant) Objects**

  Specify that an object is not modifiable

  Any attempt to modify the object is a syntax error

Example

```
//Declares a const object noon of class Time and initializes it to 12
const Time noon( 12, 0, 0 );

noon.setHour( 11 ); // error
```

```
   class Date {
   private:   int month, day; year;
              Date() { } // private default constructor
   public:    Date(int m, int d, int y) { SetDate(m, d, y); }
              void SetDate(int m, int d, int y) { month = m;  day = d; year = y;  }

              int GetMonth() const { return month; }
              int GetDay() const { return day; }
              int GetYear() const { return year; }
   };
```

The following is now valid code:

```
// although cDate is const, we can call const member functions
void PrintDate(const Date &cDate) {cout << cDate.GetMonth() << "/" <<cDate.GetDay()
<< "/" << cDate.GetYear() << endl; }

int main() {const Date cDate(10, 16, 2020);  PrintDate(cDate); return 0;}
```

MUTABLE DATA MEMBERS

- A mutable data member is **never constant**.
- It can be modified inside constant functions also.
- Prefixing the declaration of a data member with the keyword **mutable** makes it mutable.
- This is used when a data member is needed that can be modified even for constant objects.

```
// Program to demonstrate Mutable data members
class A {
        int x; //non-mutable data member
        mutable int y; //mutable data member
        public:
        void abc() const { //a constant member function
                x++; //ERROR: cannot modify a non-constant data
                //member in a constant member function
                y++; //OK: can modify a mutable data member in a constant member function
        }
        void def() { //a non-constant member function
                x++; //OK: can modify a non-constant data member
                //in a non-constant member function
                y++; //OK: can modify a mutable data member in a non-constant member function
        }
};
```

FRIENDS

- **A class can have global non-member functions and member functions of other classes as friends. Such functions can directly access the private data members of objects of the class**.
- A friend function is a **non-member function** that has special rights to access private data members of any object of the class of whom it is a friend.
  1. A friend function is prototyped within the definition of the class of which it is intended to be a friend.
  2. The prototype is prefixed with the keyword friend.
  3. Since it is a non-member function, it is defined without using the scope resolution operator.
  4. It is not called with respect to an object.

FRIEND FUNCTIONS

- A few points about the friend functions are as follows:

1. friend keyword should appear in the prototype only and not in the definition.
2. Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
3. A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.
4. The scope resolution operator should not be used while defining a friend function.

There are situations where a function that needs to access the private data members of the objects of a class, cannot be called with respect to an object of the class.

```
// A function friendly to two classes
class  ABC;//Forward declaration
class XYZ {
   int x;
public:
   void setvalue(int i){ x=i;}
   friend void max(XYZ,ABC);
};
class ABC {
   int a;
public:
   void setvalue(int i){a=i;}
   friend void max(XYZ,ABC);
};
```

```
//definition of friend
void max(XYZ m, ABC n) {
   if(m.x>=n.a)
      cout<<m.x;
   else
      cout<<n.a;
}

void main() {
   ABC abc;
   abc.setvalue(10);
   XYZ xyz;
   xyz.setvalue(20);
   max(xyz,abc);
}
Output 20
```

```
// Swapping private data of two classes
class class_2; // forward declaration
class class_1 {
   int value_1;
public:
   void indata(int a){Value_1=a;}
   void display(void){cout<<value1<<endl;}
   friend void exchange(class_1 &, class_2 &);
   //passing objects by reference
};
class class_2 {
   int value2;
public:
   void indata(int a){value2=a;}
   void display(void){ cout<<value2<<endl; }
   friend void exchange(class_1 & , class_2 &);
};
```

```
void exchange(class_1 & x , class_2 & y){
   int temp=x.value1;
   x.value1=y.value2;
   y.value2=temp;
}
void main() {
   class_1 C1;
   class_2 C2;
   C1.indata(100);
   C2.indata(200);
   cout<<”value before exchange”;
   C1.display();
   C2.display();
   exchange(C1,C2);
   cout<<”value after exchange”;
   C1.display();
   C2.display();
}
```

FRIEND CLASSES

- A class can be a friend of another class. Member functions of a friend class can access private data members of objects of the class of which it is a friend.

Example1: Declaring class B as friend of class A

```
class A {
        // all member functions of class B are friends of class A
        friend class B;
        ………….
};
```

Example2:

```
class B;
class A {
      int x;
public:
      int getx();
      // all member functions of B
      //are friends of A
      friend class B;
      ………….
};
```

```
class B {
      A *ptr;
public:
      void test_friend(int i);
};
void B::test_friend(int i) {
      ptr->x=i;
}
```

FRIEND CLASSES

Properties of friendship

- **Not symmetric** (if B a friend of A, A not necessarily a friend of B)
- **Not transitive** (if A a friend of B, B a friend of C, A not necessarily a friend of C)
- Friend Member Functions are used when one wants to make some specific member functions of one class friendly to another class.
- Any problem of **circular dependence** is solved by forward declaration i.e. forward declare a class that requires a friend.

Example3: friendship is not transitive

```
class B;
class C;
class A {
      int x;
public:
      int getx();
      friend class B;
};
class B {
      friend class C;
};
class C {
      void fun( A *ptr){
            ptr->x++; // C is not friend of A
      }
};
```

FRIEND MEMBER FUNCTIONS

- Friend Member Functions are used when one wants to make some specific member functions of one class friendly to another class.

Example 4:

- Member function of class B can be made friend of class A by declaring within class A
  - **friend void B::test_friend();**

FRIENDS AS BRIDGES

- Friend functions can be used as bridges between two classes.
- Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes.

STATIC DATA MEMBERS

- Static data members hold global data that is common to all objects of the class.
- Static data members are members of the class and not of any object of the class, that is, they are not contained inside any object.
- Prefix the declaration of a variable within the class definition with the **keyword static** to make it a static data member of the class.
- It is necessary to explicitly define a static data member outside the class to avoid an error. A statement must be written to define (allocate memory for) a static member variable.
- Making static data members private prevents any change from non-member functions as only member functions can change the values of static data members.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that have been created, as in the following example:

```
#include <iostream>
class item {
    static int count;
    int number;
 public :
    void getdata(int a) {
       number =a;
       count++;
    }
    void getcount(void) {
          cout<“count:”<<count;
    }
};
```

```
int item::count;  //definition of static data member
void main() {
   item a,b,c; // count is initialized to zero
   a.getcount();//display count
   b.getcount();
   c.getcount();
   a.getdata(100);
   b.getdata(200);
   c.getdata(300);
   cout<<“after reading the data\n”;
   a.getcount();
   b.getcount();
   c.getcount();
}
```

Output:

```
count: 0
count: 0
count: 0
After reading the data
count:  3
count:  3
count:  3
```

Note: Static members can be initialized in the program as:

```
int item::count;
```

The **type and scope** of each static data member must be defined **outside the class definition**. This is because the **static members are stored separately rather than as a part of an object.**

CONST (CONSTANT) OBJECTS AND CONST MEMBER FUNCTIONS

- **Keyword const**

  Specify that an object is not modifiable

  Any attempt to modify the object is a syntax error

- **Example**

  const Time noon(12, 0, 0);

  Declares a **const** object **noon** of class **Time** and initializes it

CONST (CONSTANT) OBJECTS AND CONST MEMBER FUNCTIONS

- **const** objects require **const** functions
- Member functions declared **const** cannot modify their object
- **const** must be specified in function prototype and definition

Prototype:

```
ReturnType FunctionName(param1,param2…) const;
```

Definition:

```
ReturnType FunctionName(param1,param2…) const { …}
```

CONST (CONSTANT) OBJECTS AND CONST MEMBER FUNCTIONS

Example:

```
int A::getValue() const
{
     return privateDataMember
};
```

- Returns the value of a data member but doesn't modify anything so is declared **const**

- Constructors / Destructors cannot be **const,** They need to initialize variables

CONST (CONSTANT) OBJECTS AND CONST MEMBER FUNCTIONS

- Only constant member functions can be called with respect to constant objects. (here ‘this’ pointer becomes the ‘constant pointer to a constant’., i.e. ,for e.g., ***`const Distance *const)`***

- Non-constant member functions cannot be called with respect to constant objects.

- Constant as well as non-constant functions can be called with respect to non-constant objects.

```
class Time {
public:
// set functions
   void setTime( int, int, int ); // set time
   void setHour( int ); // set hour
   void setMinute( int ); // set minute
   void setSecond( int ); // set second

// get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

// print functions (normally declared const)
   void printMilitary() const; // print military time
   void printStandard(); // print standard time
private:
   int hour; // 0 - 23
   int minute; // 0 - 59
   int second; // 0 - 59
};
```

```
void Time::setTime( int h, int m, int s ) {
   setHour( h );
   setMinute( m );
   setSecond( s );
}
 // Set the hour value
 void Time::setHour( int h ){ hour = ( h >= 0 && h < 24 ) ? h : 0; }
 // Set the minute value
 void Time::setMinute( int m ) { minute = ( m >= 0 && m < 60 ) ? m : 0; }
 // Set the second value
 void Time::setSecond( int s ) { second = ( s >= 0 && s < 60 ) ? s : 0; }
int Time::getHour() const { return hour; }
 // Get the minute value
 int Time::getMinute() const { return minute; }
// Get the second value
int Time::getSecond() const { return second; }
```

```
// Display military format time: HH:MM
void Time::printMilitary() const {
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"<<
   ( minute < 10 ? "0" : "" ) << minute;
}
void Time::printStandard() { // should be const
   cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":" <<
   (minute < 10 ? "0" : "")<< minute << ":" << (second < 10 ? "0" : "")
   << second << ( hour < 12 ? " AM" : " PM" );
}
```

CONST (CONSTANT) OBJECTS AND CONST MEMBER FUNCTIONS

```
int main() {
    Time wakeUp( 6, 45, 0 ); // non-constant object
    const Time noon( 12, 0, 0 ); // constant object
    // MEMBER FUNCTION OBJECT
    wakeUp.setHour( 18 ); // non-const non-const
    noon.setHour( 12 ); // non-const const
    wakeUp.getHour(); // const non-const
    noon.getMinute(); // const const
    noon.printMilitary(); // const const
    noon.printStandard(); // non-const const
    return 0;
 }
```

Static member function

- This function’s sole purpose is to access and/or modify static data members of the class.
- Prefixing the function prototype with the keyword *static* specifies it as a static member.
- ***Static member functions can access only static data members of the class.***
- A static member function can be called using the class name (instead of its objects)as follows:

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

```
#include <iostream>
class test {
    int code;
    static int count;
public:
    void setcode (void){
        code=++count;
    }
void showcode(void){
    cout <<”object no:”<<code<<endl;
}
static void showcount(void) {
    cout<<“count:”<<count<<endl;}
};
```

```
int test::count;
int main() {
    test t1,t2 ;
    t1.setcode();
    t2.setcode();
    // accessing static functions
    test::showcount();
    test t3;
    t3.setcode();
    test::showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
}
```

Output:

```
Count : 2
Count : 3
Object no: 1
Object no: 2
Object no: 3
```

Objects and Functions

- Objects can appear as local variables inside functions.
- They can also be passed by value or by reference to functions.
- They can be returned by value or by reference from functions.

```
class complex {
  float real;
  float imag;
public :
  void getdata(float x, float y) {real=x; imag=y;}
  void display(void){ cout <<real<<“+i”<<imag<<endl; }
  complex sum(complex);//declaration with objects as arguments
};
complex complex ::sum(complex x1) {
   complex t;
   t.real=real+x1.real;
   t.imag=imag+xx.imag;
   return t;
}
```

```
void main() {
   complex C1,C2,C3;
   C1.getdata(1.2,4.5);
   C2.getdata(3.1,3.3);
   C3=C1.sum(C2);
   C1.display();
   C2.display();
   C3.display();
}
```

```
Output:
1.2i+4.5
3.1i+3.3
4.3i+7.8
```

Passing objects using pass by reference

Passing objects as an argument using pass by reference method

Only address of the object is transferred to the function.

Example: Swap any two complex numbers

```
class complex {
   float real;
   float imag;
public :
  void getdata(float x, float y) {real=x;imag=y;}
   void display(void) {cout <<real<<"+i"<<imag<<endl; }
   void swap(complex &, complex &);
 / /declaration with objects as   arguments
};
```

Passing objects using pass by reference

```
void complex::swap(complex &x1,complex &x2)
{   float temp =x1.real;
    x1.real=x2.real;
    x2.real=temp;
    temp= x1.imag;
    x1.imag=x2.imag;
    x2.imag=temp;
}
```

// Returning object by reference
Complex & swap(complex &, complex &);
can be called as
Complex &c3= c1.swap(c1,c2);

```
void main() {
    complex C1,C2,C3;
    C1.getdata(1.2,4.5);
    C2.getdata(3.1,3.3);
    C3.swap(C1,C2);
    C1.display();
    C2.display();
}
Output:
3.1i+3.3
1.2i+4.5
```

Objects and Arrays

1. **Array of objects: Array of variables of the type class.**

For example:

```
void main() {
   complex C[10];
   for (int i=0;i<10; i++)
      C[i].getdata();
   for (int i=0;i<10; i++)
      C[i].display();
}
```

Objects and Arrays

Arrays inside objects

- An Array can be declared inside a class.
- Such an array becomes a member of all objects of the class.

It can be manipulated/accessed by all member functions of the class. **example:**

```
# define SIZE 5
class array
{
     int a[SIZE];
public:
    void setval(int, int);
    int getval(int);
};
```

```
void  array::setval(int p, int v) {
    if (p>=SIZE)
    return; // throw an exception
    a[p]=v;
}
int  array::getval(int p) {
    if (p>=SIZE)
    return -1; // throw an exception
    else return a[p];
}
```

NAMESPACES

- The problem: When two variables (or functions etc.) in global scope have the same identifier, we get a compile-time error.
- To avoid such **name collisions**, programmers need to use unique identifiers in their own code.
- In C, if you use multiple **third-party libraries** and there is a name collision, you have three choices:
  1. Get the source code for the libraries and modify and recompile it
  2. Ask one of the library publishers to rename their identifiers and republish the library
  3. Decide not to use one of the libraries.
- Often, none of these options is available. To tackle this problem, C++ introduced namespaces.

NAMESPACES

- All identifiers declared within a defined block are associated with the block’s namespace identifier.
- All references to these identifiers from outside the block must indicate the namespace identifier.
- One example is the namespace std, in which Standard C++ defines its library’s identifiers, such as the ***cout*** stream object.
- You can access objects in the namespace **std** in the following way using “::” operator or Or, you can use the **using namespace** statement

```
#include <iostream>
int main(){
   std::cout << “Hello World!”;
   return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
   cout << “Hello World!”;
   return 0;
}
```

NAMESPACES

- This is how you define your own namespaces:

```
#include <iostream>
namespace MyNames {
     int value1 = 10;
     int value2 = 20;
     int ComputeSum() {
           return (value1 + value2);
     }
}
int main(){
     std::cout << MyNames::ComputeSum() << std::endl;
}
```

NAMESPACES

- If you use multiple **using namespace** statements, you may get a compile-time error due to ambiguity:

```
#include <iostream>
namespace MyNames
{
        int value1 = 10;
        int value2 = 20;
}
namespace MyOtherNames
{
        int value1 = 30;
        int value2 = 40;
}
```

```
using namespace std;
using namespace MyNames;
using namespace MyOtherNames;

int main()
{
    value1 = 50;
    cout<<value1;
}
```

NAMESPACES

- **You can also define nested namespaces:**

```
#include <iostream>
namespace MyNames {
      int value1 = 10;
      int value2 = 20;
      namespace MyInnerNames {
            int value3 = 30;
      }
}
int main() {
      std::cout << MyNames::value1 << std::endl;
      std::cout << MyNames::MyInnerNames::value3 << std::endl;
}
```

NAMESPACES

- **Problem**: Some namespaces have long names. Qualifying the name of a class that is enclosed within such a namespace, with the name of the namespace, is cumbersome.

```
namespace a_very_very_long_name {
      int value3;
}
//Using  namespace a_very_very_long_name;
void main() {
      std::cout<<a_very_very_long_name ::value3 << std::endl;
}
```

- **Solution**: Assigning a short alias to such a long namespace name solves the problem.

```
namespace  x=a_very_very_long_name ; //declaring alias
void main(){
      std::cout<<x::value3 << std::endl;
}
```

NESTED CLASSES

- A class can be defined inside another class. Such a class is known as a **nested class**.
- The class that contains the nested class is known as the ***enclosing class.***
- Nested classes can be defined in the private, protected, or public portions of the enclosing class.

Example1

```
class A {
  class B {
        /* definition of class B*/
  };
  /* definition of class A*/
};
```

NESTED CLASSES

- A nested class is created if it does not have any relevance outside its enclosing class. By defining the class as a nested class, we avoid a name collision.
- The size of objects of an *enclosing class* is not affected by the presence of nested classes.

Example2:

```
class A {
    int x;
Public:
    class B {
          int y;
    };
};
void main() {
    cout << sizeof(int)<<“\t”<<sizeof(A)<<endl;
}
Output: 4  4
```

NESTED CLASSES

- **Member functions of a nested class** can be defined outside the definition of the enclosing class by prefixing the function name with the name of the enclosing class followed by the scope resolution operator. This, in turn, is followed by the name of the nested class followed again by the scope resolution operator.

Example3:

```
class A {
   public:
         class B {
         public:
                  void Btest();
         };
   };
   void A::B::Btest() {
         /* definition */
   }
```

NESTED CLASSES

- **A nested class** may be only prototypes within its enclosing class and defied later.
- The name of the enclosing class followed by the scope resolution operator is required

Example3:

```
class A {
        class B; //prototype only
};
class A::B {
        // class definition
};
```

- The objects of the nested class are defined outside the member functions of the enclosing class as:

A::B B1; //will compile only if class B is defined within the public section of class A.

NESTED CLASSES

- **An object of the nested class** can be used in any of the member functions of the enclosing class without the scope resolution operator.
- Object of the nested class can be a member of the enclosing class. And only the public members of the object can be accessed.

Example3:

```
class A {
        class B {
        public:
                void Btest();
        };
        B B1;
        public:
        void Atest();
};
void A::Atest() {
        B1.Btest();
        B B2; B2.Btest();
}
```

NESTED CLASSES

- **Member functions of the nested class can access the non-static members** of the enclosing class through an object, a pointer or a reference only

Example3:

```
class A {
Public:
     void Atest();
     class B {
     public:
            void Btest1(A &);
            void Btest2();
     };
};
void A::B::Btest1(A & Aref) {  Aref.Atest();  //ok }
void A::B::Btest2() {   Atest();  //Error }
```

CHAPTER 4
CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS AND DESTRUCTORS

- Constructors – introduction and features
- The zero-argument constructor
- Parameterized constructors
- Creating a parameterized constructor for the class *String*
- Explicit constructors
- Copy constructor
- Destructors

CONSTRUCTORS

- The constructor gets called automatically for each object that has just got created.
- It appears as member function of each class, whether it is defined or not.
- It has the same name as that of the class.
- It may or may not take parameters.
- It does not return anything (not even void).
- Constructors fulfill the need for a function that guarantees initialization of member data of a class.
- Domain constraints on the values of data members can also be implemented via constructors.
- The constructor gets called automatically for each object when it is created.
- The prototype of a constructor is `<class name> (<parameter list>);`

Example1:

```
    class A {  /* class definition*/ };
    void main() {  A A1; }
```

CONSTRUCTORS

- **Constructors do not actually allocate memory** for objects. They are member functions that are called for each object immediately after memory has been allocated for the object.

Example 2:

```
class A {
    int x;
public:
    void setx(int);
    int getx();
};
void main()  {  A  A1; // object declared  }
```

- The statement in main is transformed to

```
A  A1;  // 4 bytes of memory  is allocated for the object
A1.A();// constructor is called implicitly by the compiler
```

- It is forbidden to call the constructor explicitly for an existing object.

CONSTRUCTOR

Example 3:

1. Before (without constructor):

```
class A {  /* class definition*/  };
```

2. After (with constructor):

```
class A  {
   public:
   A(); //prototype inserted implicitly by the compiler
};
A::A() {
   // empty definition inserted implicitly by the compiler
}
```

THE ZERO-ARGUMENT CONSTRUCTOR

- The constructor is a **non-static member function.**
- It is called for an object. It, therefore, takes **the *this* pointer as a leading formal argument.**
- The address of the invoking object is passed as a leading parameter to the constructor call.
- This means that the members of the invoking object can be accessed from within the definition of the constructor.
- The constructor gets called for each object when the object is created.

THE ZERO-ARGUMENT CONSTRUCTOR

- The constructor that does not take any arguments and is called the zero-argument constructor.

Example 4:

```
class A {
    int x;
public:
    A(); //our own constructor
    void setx(int);
    int getx();
};
A::A() { cout<<“ constructor of class A called\t”; }
void main() {
    A A1;
    cout<<“program ends\n”;
}
Output : constructor of class A called      program ends
```

THE ZERO-ARGUMENT CONSTRUCTOR

- A user-defined constructor implements domain constraints on the data members of a class.

Example 5:

```
class distance {
    int inch;
    int feet;
public:
    distance(); //our own constructor
    /* rest of the class definition*/
};
distance::distance() { inch=0;  feet=0;  }
void main()  {
    distance d1;
    cout<<d1.getfeet()<<"\t"<<d1.getinch();
}
Output : 0 0
```

THE ZERO-ARGUMENT CONSTRUCTOR

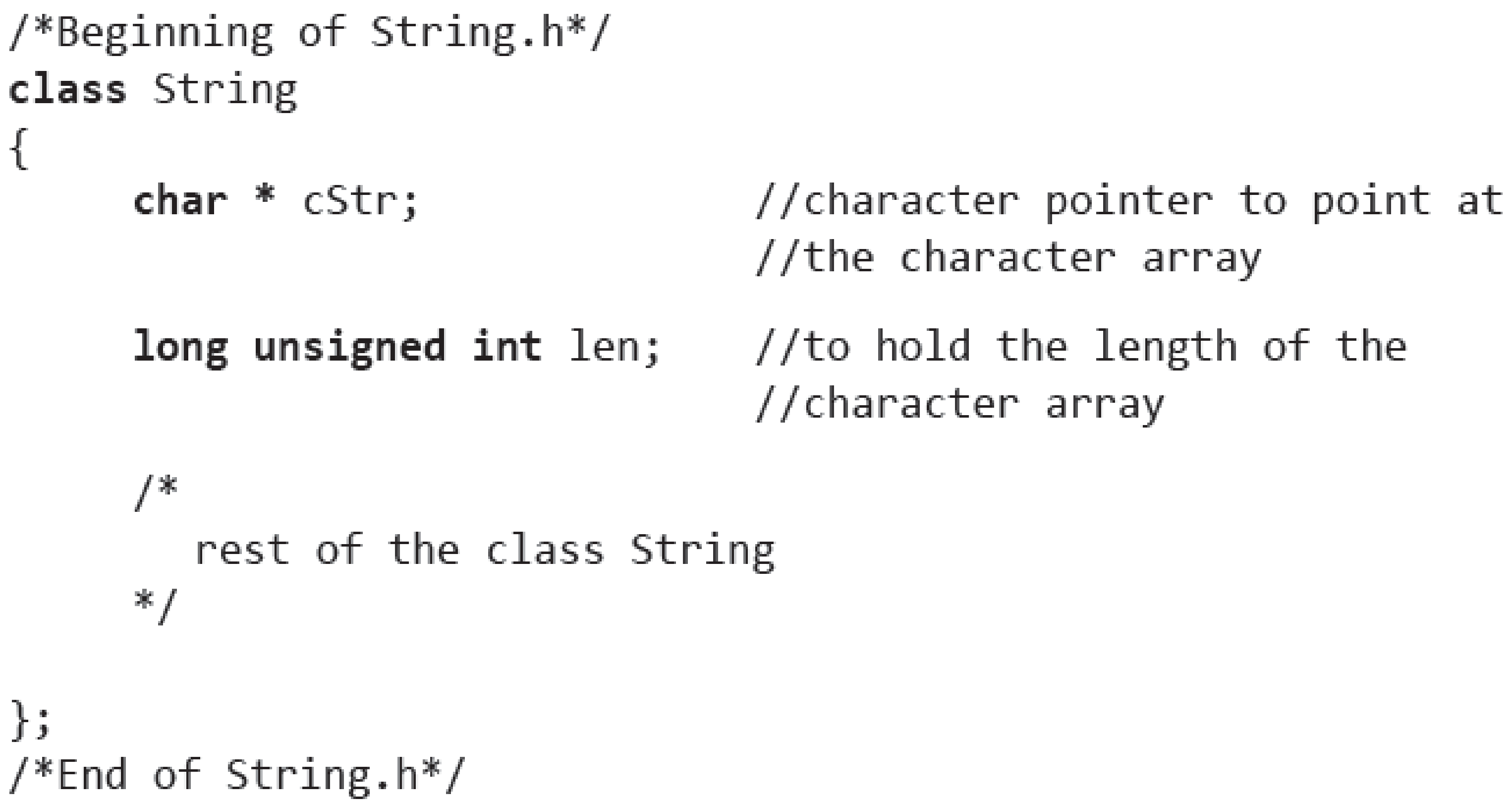
- The constructor provided by default by the compiler also does not take any arguments.
- The terms 'zero-argument constructor' and 'default constructor' are used interchangeably.

Running example of class *String*

- It will have two data members. Both these data members will be private.
- The first data member will be a character pointer. It will point at a dynamically allocated block of memory that contains the actual character array.
- The other data member will be a long unsigned integer that will contain the length of this character array.

THE ZERO-ARGUMENT CONSTRUCTOR

Example 6



```
/*Beginning of String.h*/
class String
{
     char * cStr;                   //character pointer to point at
                                    //the character array

     long unsigned int len;         //to hold the length of the
                                    //character array

     /*
       rest of the class String
     */

};
/*End of String.h*/
```

THE ZERO-ARGUMENT CONSTRUCTOR

- Suppose ‘S1’ is an object of the class *String* and string ‘abc’ has been assigned to it. The address of the first byte of the memory block containing the string is stored in the ‘cStr’ portion of ‘S1’.

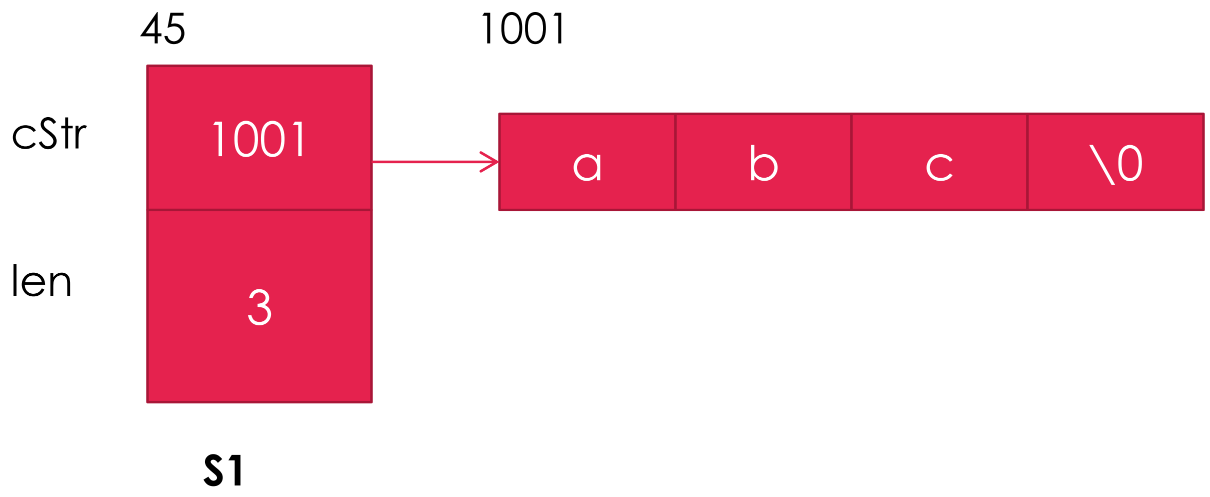


Figure: Memory layout of an object of the class ‘String’

The following two conditions should be implemented on all objects of the class *String*.

- ‘cStr’ should either point at a dynamically allocated block of memory exclusively allocated for it (i.e., no other pointer should point at the block of memory being pointed at by ‘cStr’) or ‘
- cStr’ should be NULL.

THE ZERO-ARGUMENT CONSTRUCTOR

- When an object of the class ‘String’ is created , the ‘cStr ‘ portion of the object should be initially set to NULL (and ‘len’ should be set to 0)
- The prototype and the definition of the constructor are as shown in example 7

Example 7:

```
class String {
  char *Cstr;
  unsigned int len;
public:
  String(); //our own constructor
  /* rest of the class definition*/
};
String::String() {
  Cstr=NULL;
  len=0;
}
void main() {  String S1;   }
```

PARAMETERIZED CONSTRUCTORS

- Constructors take arguments and can, therefore, be overloaded.
- A user-defined parameterized constructor can also be called by creating an object in the heap.
- The parameterized constructor is prototyped and defined just like any other member function except for the fact that it does not return any value.
- **If the parameterized constructor is provided and the zero-argument constructor is not provided, the compiler will not provide the *default constructor*.**
- Default values given to parameters of a parameterized constructor make the zero- argument constructor unnecessary.

Example 8:

```
class distance {
      int inch;
      int feet;
public:
      distance( int=0, int=0); //default values given
      /* rest of the class definition*/
};
distance D1; //assigns inch and feet of D1 with 0, no need of zero argument constructor
```

PARAMETERIZED CONSTRUCTORS

Example 9

```
class String  {
    char *Cstr;
    unsigned int len;
public:
    String(); //our own constructor
    String (char * P);
    char * getstring() { return Cstr;}
    /* rest of the class definition*/
};
```

```
String::String() {
    Cstr=NULL;
    len=0;
}
String::String(char *str) {
    len= strlen(str);
    Cstr=new char[len+1];
    strcpy(Cstr,str);
}
void main() {
    String S1;
}
```

PARAMETERIZED CONSTRUCTORS

- Here, a value is assigned for the argument of the parameterized constructor.
- The constructor would handle the following statements:

1. String s1(“abc”);

OR

2. char * cPtr = “abc”;

String s1(cPtr);

OR

3. char cArr[10] = “abc”;

String s1(cArr);

- In each of these statements, we are essentially passing the base address of the memory block in which the string itself is stored to the constructor.

COPY CONSTRUCTOR

- The copy constructor is a special type of parameterized constructor which copies one object to another.
- It is called when an object is created and equated to an existing object at the same time.
- The copy constructor is called for the object being created. The pre existing object is passed as a parameter to it.
- The copy constructor member-wise copies the object passed as a parameter to it into the object for which it is called.
- If the copy constructor for a class is not defined, the compiler defines it for us. But in either case, it is called under the following three circumstances:
    1. When an object is created and simultaneously equated to another existing object, the copy constructor is called for the object being created. The object to which this object was equated is passed as a parameter to the copy constructor.

Example 10:

```
A A1;
A A2=A1; //copy constructor called
or
A A2(A1)
or
A *ptr = new A(A1);
```

COPY CONSTRUCTOR

2. When an object is created as a non-reference formal argument of a function. The copy constructor is called for the argument object. The object passed as a parameter to the function is passed as a parameter to the copy constructor.

Example 11:

```
void abc(A);
A A1;
abc(A1); //copy constructor called
void abc(A A2) {
 // definition of abc()
}
```

COPY CONSTRUCTOR

3. When an object is created and simultaneously equated to a call to a function that returns an object. The copy constructor is called for the object that is equated to the function call. The object returned from the function is passed as a parameter to the constructor.

Example 12:

```
A abc()
{
        A A1;
        //remaining definition of abc
        return A1;
}
A A2=abc(); //copy constructor called
```

COPY CONSTRUCTOR

- Default copy constructor is defined by the compiler . Here the formal parameter is a reference, due to this no separate memory is allocated
- The prototype and the definition of the default copy constructor defined by the compiler are as follows: **Example 13:**

```
Class A {
    public:
    A (A&); //default copy constructor
};
A::A(A & Aobj)  {
    *this=Aobj;
}
```

Now the statement:

```
    A A2=A1;
```

is converted as follows:

```
    A A2;
    A2.A(A1);    //copy constructor is called for A2
```

COPY CONSTRUCTOR

Example 14

```
class String {
    char *Cstr;
    unsigned int len;
public:
    String( String &); //our own copy constructor
    /* rest of the class definition*/
};
String::String(String &ss) {
    if (ss.CStr==NULL) {
        Cstr=NULL;  len=0;
    } else {
        len= strlen(ss.str);
        Cstr=new char[len+1]; //dynamically allocate a separate memory block and copy
        strcpy(Cstr, ss.Cstr);
    }
}
void main() {
    String S1(“abc”);  String s2=s1;
}
```

DESTRUCTOR

- This function is the opposite of the constructor in the sense that it is invoked when an object ceases to exist. The definition of a destructor must obey the following rules:
- The destructor has the same name as the class but its name is prefixed by a **tilde(~)**.
- The destructor has no arguments and no a return value.
- Destructors cannot be overloaded.

The destructor for the class Person is thus declared as follows:

```
class Person    {
   public:
   Person();  // constructor
   ~Person(); // destructor
};
```

THE DESTRUCTOR

The position of the constructor(s) and destructor in the class definition is dictated by following convention:

- First the constructors are declared, then the destructor, and only then other members are declared.
- The main task of a destructor is to make sure that memory allocated by the object (e.g., by its constructor) is properly deleted when the object goes out of scope.

IMPLEMENTATION OF DESTRUCTOR

Output

```
Enter main
No of obj created 1
No of obj created 2
No of obj created 3
No of obj created 4
Enter block1
No of objects created 5
No of objects destroyed 5
Re enter main
No of objects destroyed 4
No of objects destroyed 3
No of objects destroyed 2
No of objects destroyed 1
```

```
int count=0;
class alpha {
    alpha(){
        count++;
        cout<<”no of objects created ”<<count;
    }
    ~alpha() {
        cout<<”no of objects destroyed ”<<count;
        count--;
    }
};
void main() {
    cout<< “enter main”
    alpha A1,A2,A3,A4;
    {
        cout<<”Enter block1”
        alpha A5;
    }
    cout<<”Re enter main”;
}
```

IMPLEMENTATION OF DESTRUCTOR

- Let s1 be an object of class String. If s1.Cstr is dynamically allocated in the heap area. After s1 gets destroyed, this memory block remains allocated as a locked up lost resource. This allocated memory, could be deallocated using delete operator.

Example 14

```
class String
{
    char *Cstr;
    unsigned int len;
public:
    ~String( ); //our own destructor
    /* rest of the class definition*/
};
String::~String()
{
    if (Cstr!=NULL)
    delete [] Cstr; //if memory exists destroy it
}
```

THANK YOU!