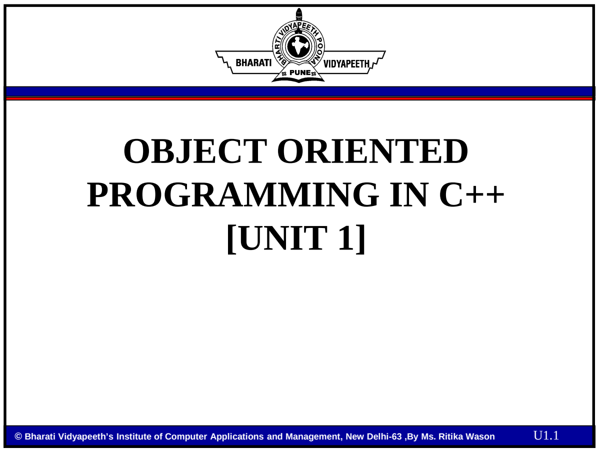
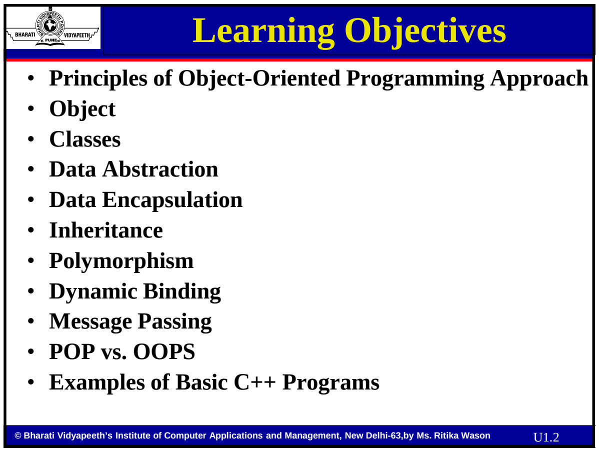


OBJECT ORIENTED PROGRAMMING IN C++ [UNIT 1]

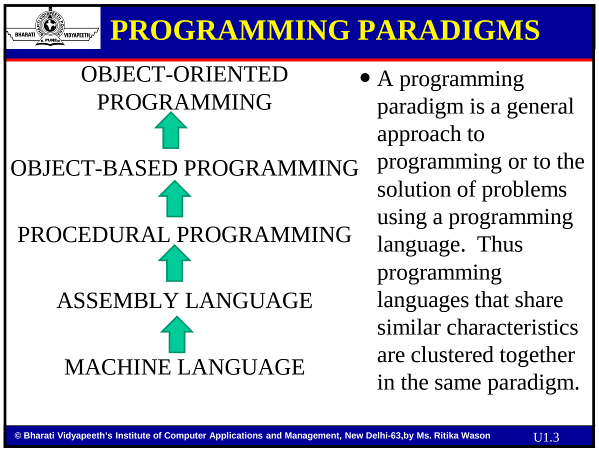
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63 ,By Ms. Ritika Wason U1.1



Learning Objectives

- Principles of Object-Oriented Programming Approach
- Object
- Classes
- Data Abstraction
- Data Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing
- POP vs. OOPS
- Examples of Basic C++ Programs

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.2



PROGRAMMING PARADIGMS

OBJECT-ORIENTED PROGRAMMING

OBJECT-BASED PROGRAMMING

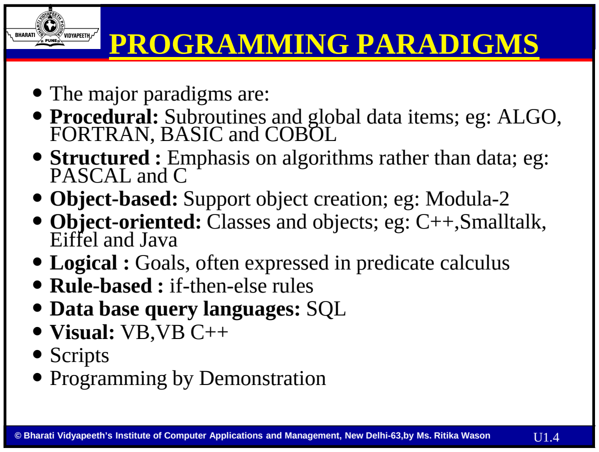
PROCEDURAL PROGRAMMING

ASSEMBLY LANGUAGE

MACHINE LANGUAGE

- A programming paradigm is a general approach to programming or to the solution of problems using a programming language. Thus programming languages that share similar characteristics are clustered together in the same paradigm.

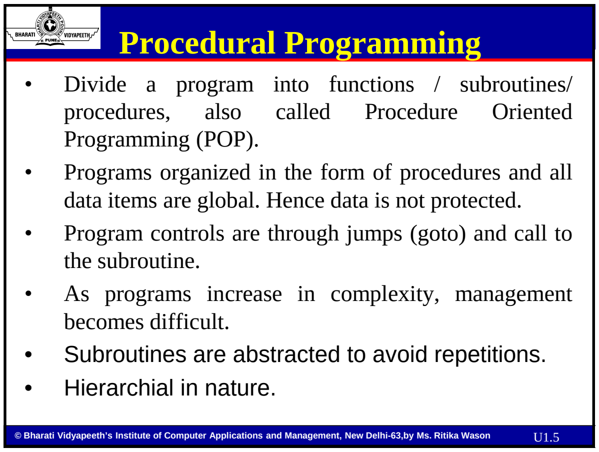
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.3



PROGRAMMING PARADIGMS

- The major paradigms are:
- **Procedural:** Subroutines and global data items; eg: ALGO, FORTRAN, BASIC and COBOL
- **Structured :** Emphasis on algorithms rather than data; eg: PASCAL and C
- **Object-based:** Support object creation; eg: Modula-2
- **Object-oriented:** Classes and objects; eg: C++,Smalltalk, Eiffel and Java
- **Logical :** Goals, often expressed in predicate calculus
- **Rule-based :** if-then-else rules
- **Data base query languages:** SQL
- **Visual:** VB,VB C++
- Scripts
- Programming by Demonstration

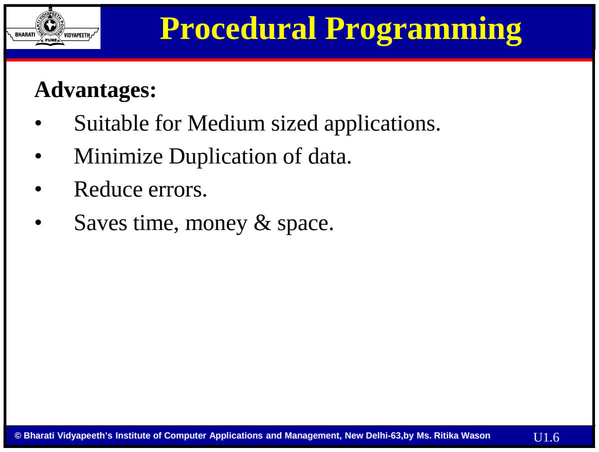
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.4



Procedural Programming

- Divide a program into functions / subroutines/ procedures, also called Procedure Oriented Programming (POP).
- Programs organized in the form of procedures and all data items are global. Hence data is not protected.
- Program controls are through jumps (goto) and call to the subroutine.
- As programs increase in complexity, management becomes difficult.
- Subroutines are abstracted to avoid repetitions.
- Hierarchial in nature.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.5

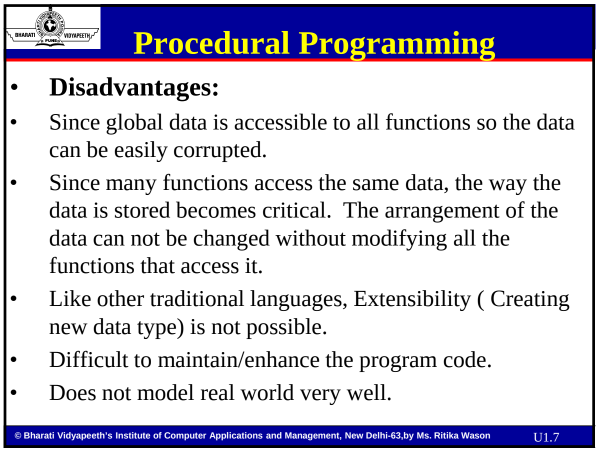


Procedural Programming

Advantages:

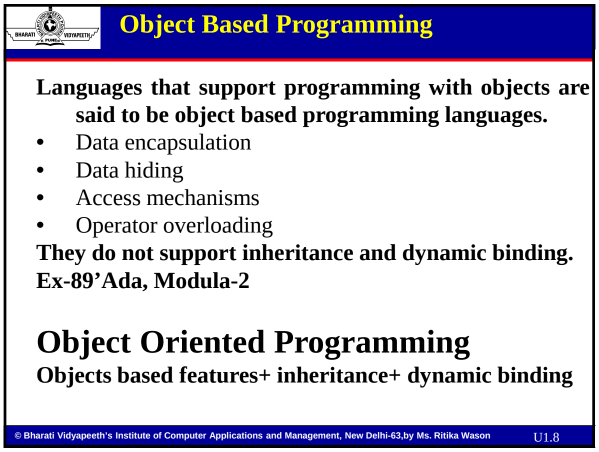
- Suitable for Medium sized applications.
- Minimize Duplication of data.
- Reduce errors.
- Saves time, money & space.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.6



Procedural Programming

- **Disadvantages:**
- Since global data is accessible to all functions so the data can be easily corrupted.
- Since many functions access the same data, the way the data is stored becomes critical. The arrangement of the data can not be changed without modifying all the functions that access it.
- Like other traditional languages, Extensibility (Creating new data type) is not possible.
- Difficult to maintain/enhance the program code.
- Does not model real world very well.



Object Based Programming

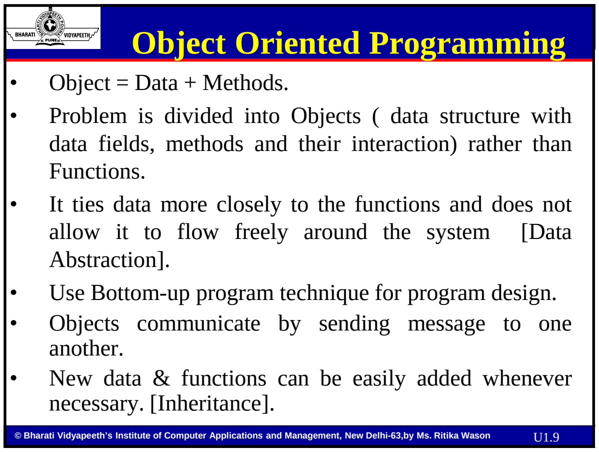
Languages that support programming with objects are said to be object based programming languages.

- Data encapsulation
- Data hiding
- Access mechanisms
- Operator overloading

They do not support inheritance and dynamic binding.
Ex-89'Ada, Modula-2

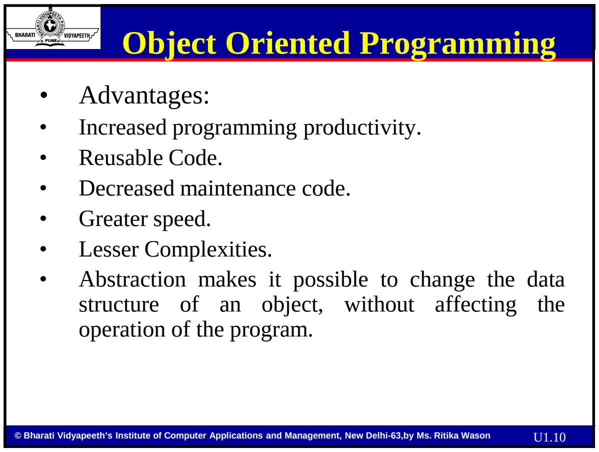
Object Oriented Programming

Objects based features+ inheritance+ dynamic binding



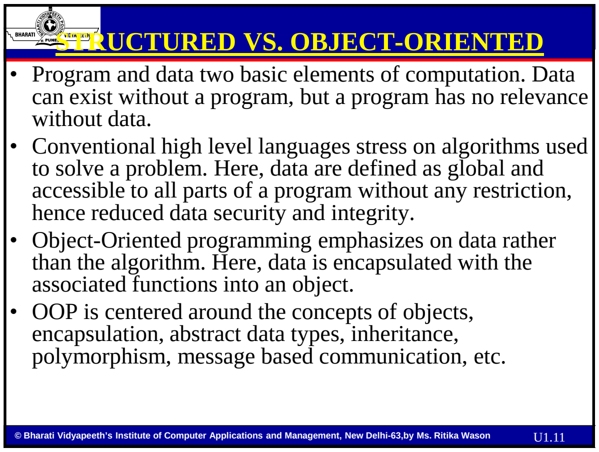
Object Oriented Programming

- Object = Data + Methods.
- Problem is divided into Objects (data structure with data fields, methods and their interaction) rather than Functions.
- It ties data more closely to the functions and does not allow it to flow freely around the system [Data Abstraction].
- Use Bottom-up program technique for program design.
- Objects communicate by sending message to one another.
- New data & functions can be easily added whenever necessary. [Inheritance].



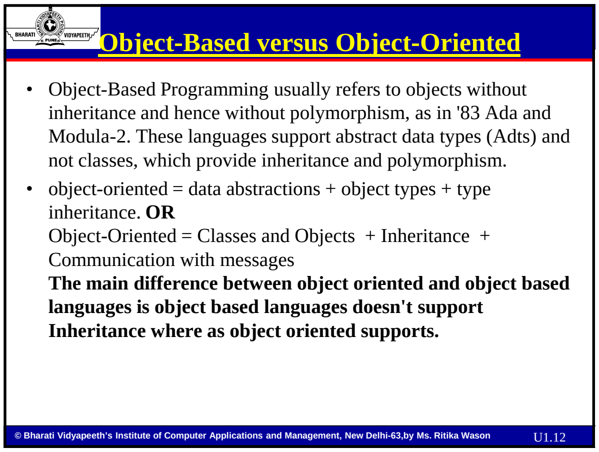
Object Oriented Programming

- Advantages:
- Increased programming productivity.
- Reusable Code.
- Decreased maintenance code.
- Greater speed.
- Lesser Complexities.
- Abstraction makes it possible to change the data structure of an object, without affecting the operation of the program.



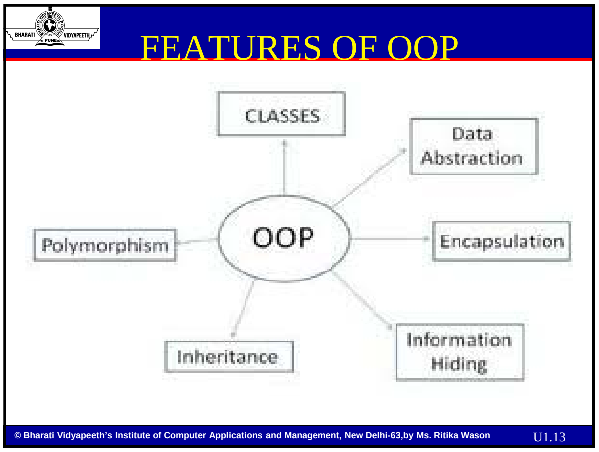
STRUCTURED VS. OBJECT-ORIENTED

- Program and data two basic elements of computation. Data can exist without a program, but a program has no relevance without data.
- Conventional high level languages stress on algorithms used to solve a problem. Here, data are defined as global and accessible to all parts of a program without any restriction, hence reduced data security and integrity.
- Object-Oriented programming emphasizes on data rather than the algorithm. Here, data is encapsulated with the associated functions into an object.
- OOP is centered around the concepts of objects, encapsulation, abstract data types, inheritance, polymorphism, message based communication, etc.

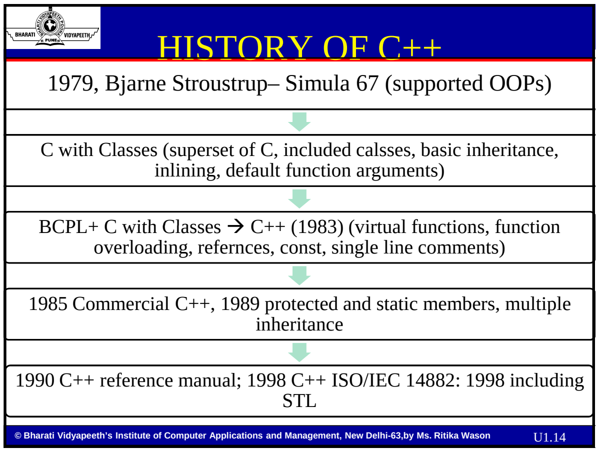


Object-Based versus Object-Oriented

- Object-Based Programming usually refers to objects without inheritance and hence without polymorphism, as in '83 Ada and Modula-2. These languages support abstract data types (Adts) and not classes, which provide inheritance and polymorphism.
- object-oriented = data abstractions + object types + type inheritance. **OR**
  Object-Oriented = Classes and Objects + Inheritance + Communication with messages
  The main difference between object oriented and object based languages is object based languages doesn't support Inheritance where as object oriented supports.



FEATURES OF OOP



HISTORY OF C++

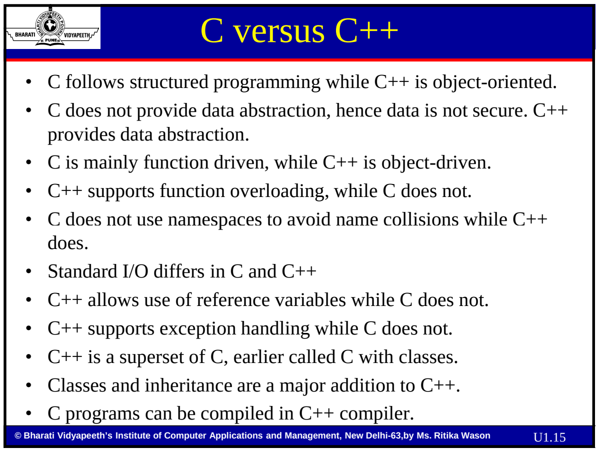
1979, Bjarne Stroustrup– Simula 67 (supported OOPs)

C with Classes (superset of C, included calsses, basic inheritance, inlining, default function arguments)

BCPL+ C with Classes → C++ (1983) (virtual functions, function overloading, refernces, const, single line comments)

1985 Commercial C++, 1989 protected and static members, multiple inheritance

1990 C++ reference manual; 1998 C++ ISO/IEC 14882: 1998 including STL



C versus C++

- C follows structured programming while C++ is object-oriented.
- C does not provide data abstraction, hence data is not secure. C++ provides data abstraction.
- C is mainly function driven, while C++ is object-driven.
- C++ supports function overloading, while C does not.
- C does not use namespaces to avoid name collisions while C++ does.
- Standard I/O differs in C and C++
- C++ allows use of reference variables while C does not.
- C++ supports exception handling while C does not.
- C++ is a superset of C, earlier called C with classes.
- Classes and inheritance are a major addition to C++.
- C programs can be compiled in C++ compiler.

Structure of C++ Program

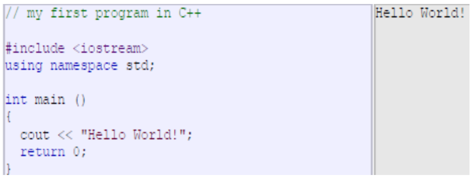
- C programs use the file extension **.C and C++ programs use the extension .CPP. A C++ compiler uses** the file extension to determine what type of program it is compiling.

Using C++ hello.cpp

```
//hello.cpp: printing Hello World message
#include <iostream.h>
void main(){
        cout<< “Hello world”;
}
```

Structure of C++ Program

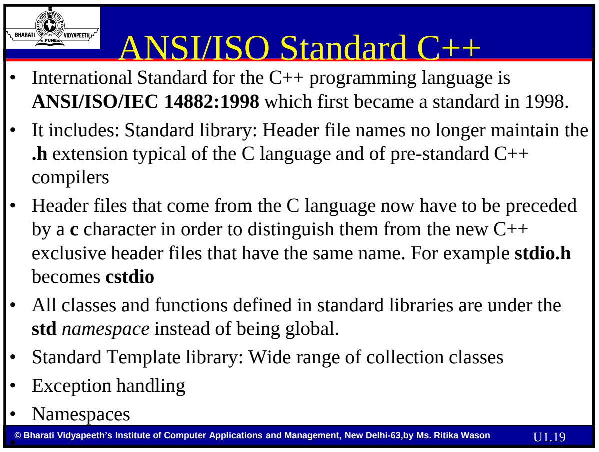
```
/* greeting.cpp greets its user. * * Input:  The name of the user * Output:
A personalized greeting */
#include <iostream.h> //preprocessor directive for iostream header file
 #include <string> //preprocessor directives
int main()//main function declaration
{ //block scope
  cout << "Please enter your first name: ";
   string firstName;
  cin >> firstName;
  cout << "\nWelcome to the world of C++, "
<< firstName << "!\n";   return 0;}
```



```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

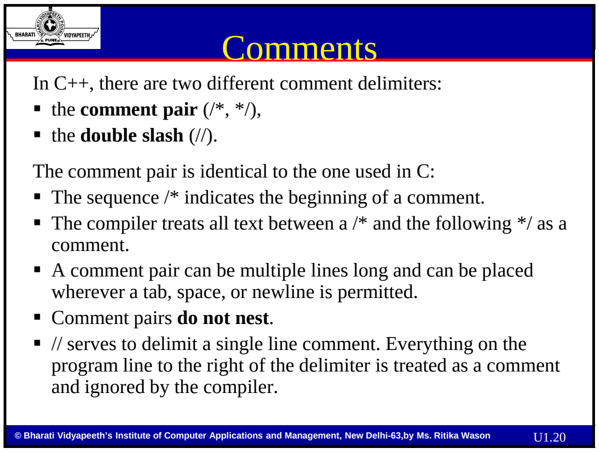
```
Hello World!
```

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std. So in order to access its functionality we declare with this expression that* we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes



ANSI/ISO Standard C++

- International Standard for the C++ programming language is **ANSI/ISO/IEC 14882:1998** which first became a standard in 1998.
- It includes: Standard library: Header file names no longer maintain the **.h** extension typical of the C language and of pre-standard C++ compilers
- Header files that come from the C language now have to be preceded by a **c** character in order to distinguish them from the new C++ exclusive header files that have the same name. For example **stdio.h** becomes **cstdio**
- All classes and functions defined in standard libraries are under the **std** *namespace* instead of being global.
- Standard Template library: Wide range of collection classes
- Exception handling
- Namespaces



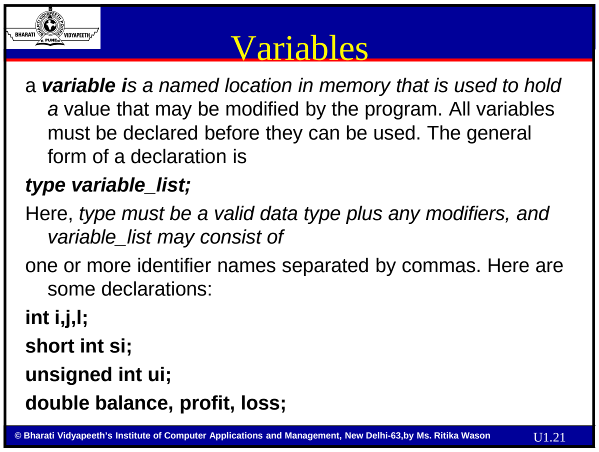
Comments

In C++, there are two different comment delimiters:

- the **comment pair** (/*, */),
- the **double slash** (//).

The comment pair is identical to the one used in C:

- The sequence /* indicates the beginning of a comment.
- The compiler treats all text between a /* and the following */ as a comment.
- A comment pair can be multiple lines long and can be placed wherever a tab, space, or newline is permitted.
- Comment pairs **do not nest**.
- // serves to delimit a single line comment. Everything on the program line to the right of the delimiter is treated as a comment and ignored by the compiler.



Variables

a ***variable*** *is a named location in memory that is used to hold a* value that may be modified by the program. All variables must be declared before they can be used. The general form of a declaration is

type variable_list;

Here, *type must be a valid data type plus any modifiers, and variable_list may consist of*

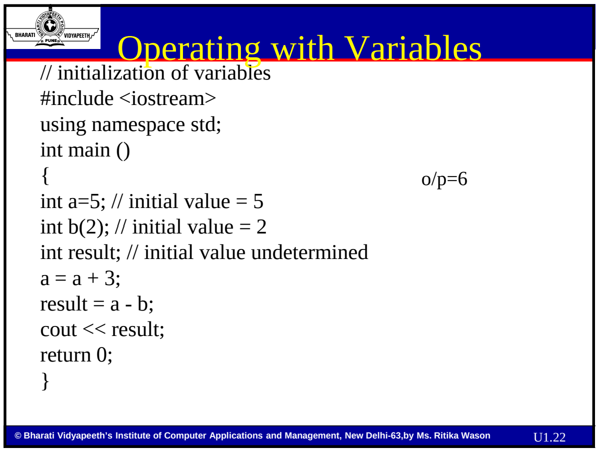
one or more identifier names separated by commas. Here are some declarations:

int i,j,l;

short int si;

unsigned int ui;

double balance, profit, loss;

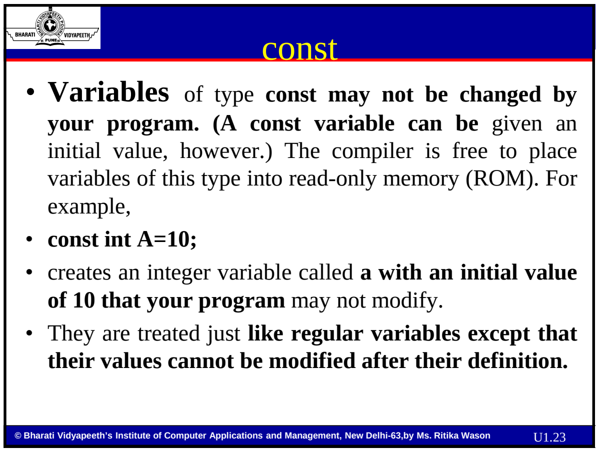


Operating with Variables

```
// initialization of variables
#include <iostream>
using namespace std;
int main ()
{
int a=5; // initial value = 5
int b(2); // initial value = 2
int result; // initial value undetermined
a = a + 3;
result = a - b;
cout << result;
return 0;
}
```

o/p=6

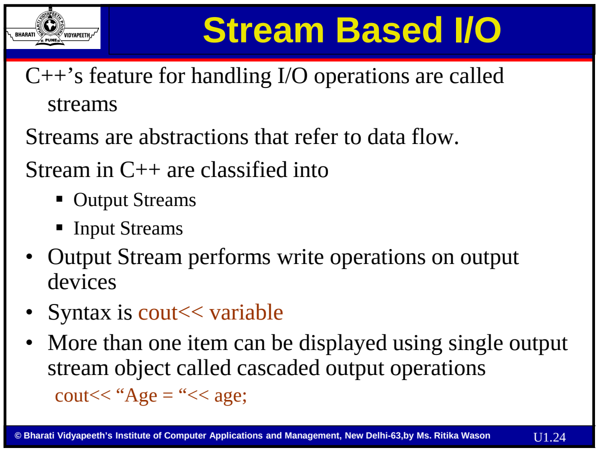
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.22



const

- **Variables** of type **const may not be changed by your program. (A const variable can be** given an initial value, however.) The compiler is free to place variables of this type into read-only memory (ROM). For example,
- **const int A=10;**
- creates an integer variable called **a with an initial value of 10 that your program** may not modify.
- They are treated just **like regular variables except that their values cannot be modified after their definition.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.23



Stream Based I/O

C++'s feature for handling I/O operations are called streams

Streams are abstractions that refer to data flow.

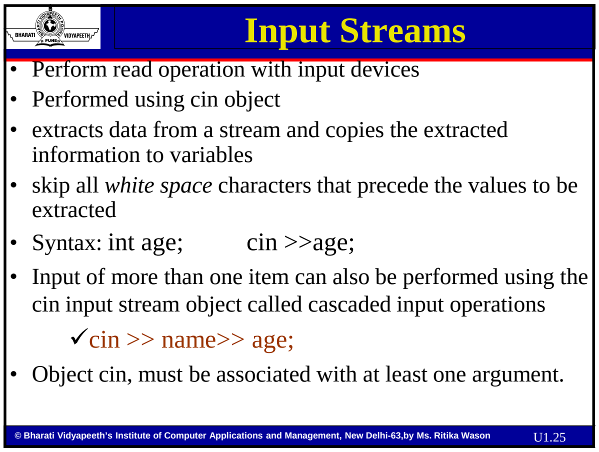
Stream in C++ are classified into

  - Output Streams
  - Input Streams

- Output Stream performs write operations on output devices
- Syntax is cout<< variable
- More than one item can be displayed using single output stream object called cascaded output operations

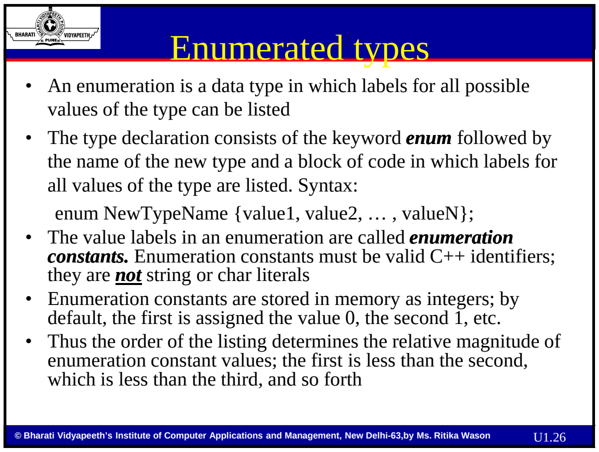
  cout<< "Age = "<< age;

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.24



Input Streams

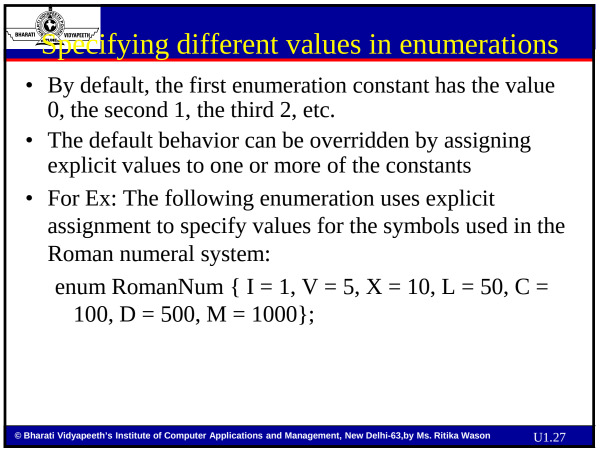
- Perform read operation with input devices
- Performed using cin object
- extracts data from a stream and copies the extracted information to variables
- skip all *white space* characters that precede the values to be extracted
- Syntax: int age;   cin >>age;
- Input of more than one item can also be performed using the cin input stream object called cascaded input operations
  - ✓cin >> name>> age;
- Object cin, must be associated with at least one argument.



Enumerated types

- An enumeration is a data type in which labels for all possible values of the type can be listed
- The type declaration consists of the keyword ***enum*** followed by the name of the new type and a block of code in which labels for all values of the type are listed. Syntax:

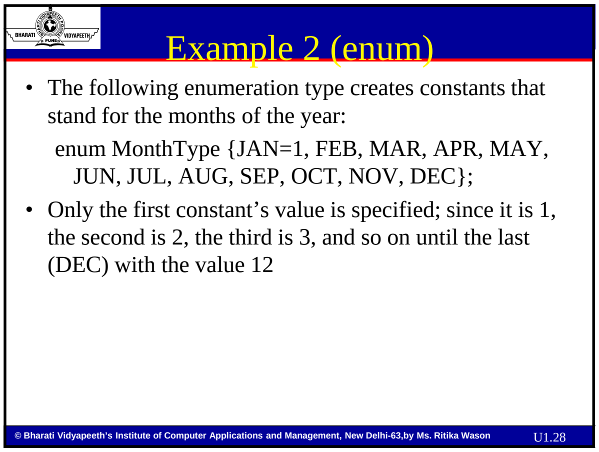
  enum NewTypeName {value1, value2, … , valueN};
- The value labels in an enumeration are called ***enumeration constants.*** Enumeration constants must be valid C++ identifiers; they are ***not*** string or char literals
- Enumeration constants are stored in memory as integers; by default, the first is assigned the value 0, the second 1, etc.
- Thus the order of the listing determines the relative magnitude of enumeration constant values; the first is less than the second, which is less than the third, and so forth



Specifying different values in enumerations

- By default, the first enumeration constant has the value 0, the second 1, the third 2, etc.
- The default behavior can be overridden by assigning explicit values to one or more of the constants
- For Ex: The following enumeration uses explicit assignment to specify values for the symbols used in the Roman numeral system:

  enum RomanNum { I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000};

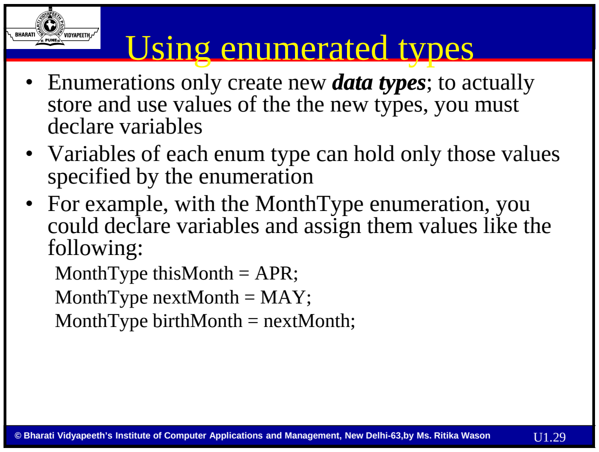


Example 2 (enum)

- The following enumeration type creates constants that stand for the months of the year:

  enum MonthType {JAN=1, FEB, MAR, APR, MAY,
     JUN, JUL, AUG, SEP, OCT, NOV, DEC};

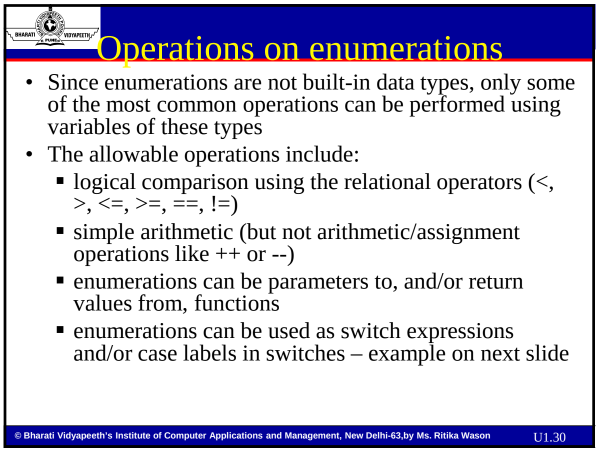
- Only the first constant's value is specified; since it is 1, the second is 2, the third is 3, and so on until the last (DEC) with the value 12



Using enumerated types

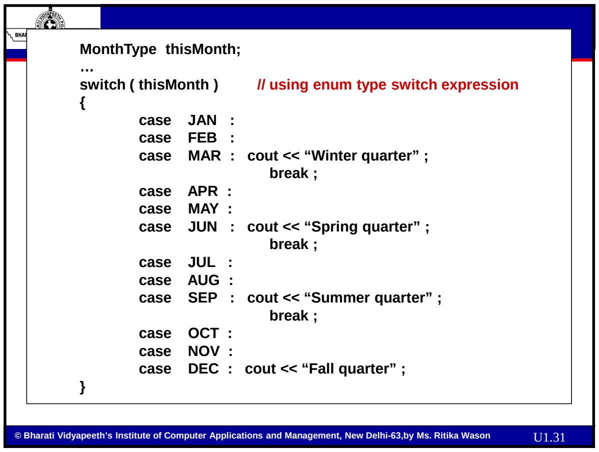
- Enumerations only create new ***data types***; to actually store and use values of the the new types, you must declare variables
- Variables of each enum type can hold only those values specified by the enumeration
- For example, with the MonthType enumeration, you could declare variables and assign them values like the following:

  MonthType thisMonth = APR;
  MonthType nextMonth = MAY;
  MonthType birthMonth = nextMonth;



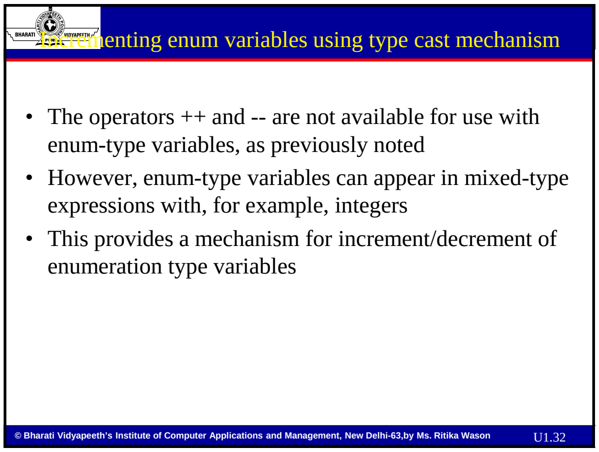
Operations on enumerations

- Since enumerations are not built-in data types, only some of the most common operations can be performed using variables of these types
- The allowable operations include:
  - logical comparison using the relational operators (<, >, <=, >=, ==, !=)
  - simple arithmetic (but not arithmetic/assignment operations like ++ or --)
  - enumerations can be parameters to, and/or return values from, functions
  - enumerations can be used as switch expressions and/or case labels in switches – example on next slide



```
MonthType  thisMonth;
…
switch ( thisMonth )        // using enum type switch expression
{
          case   JAN   :
          case   FEB   :
          case   MAR  :   cout << “Winter quarter” ;
                              break ;
          case   APR  :
          case   MAY  :
          case   JUN   :   cout << “Spring quarter” ;
                              break ;
          case   JUL   :
          case   AUG  :
          case   SEP   :   cout << “Summer quarter” ;
                              break ;
          case   OCT  :
          case   NOV  :
          case   DEC  :  cout << “Fall quarter” ;
}
```

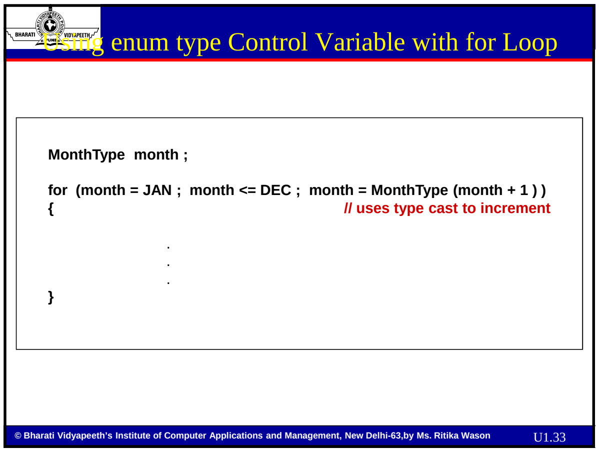
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.31



Incrementing enum variables using type cast mechanism

- The operators ++ and -- are not available for use with enum-type variables, as previously noted
- However, enum-type variables can appear in mixed-type expressions with, for example, integers
- This provides a mechanism for increment/decrement of enumeration type variables

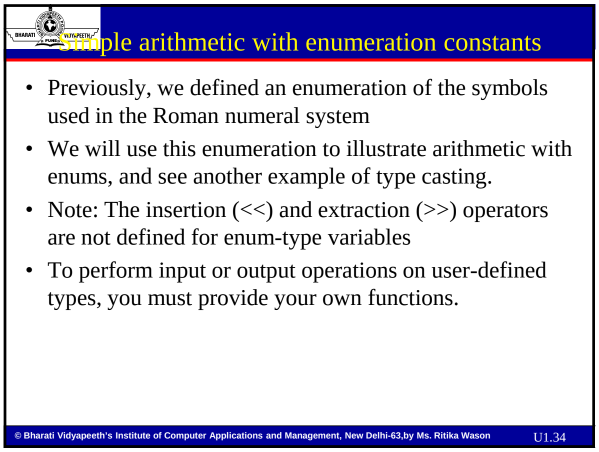
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.32



Using enum type Control Variable with for Loop

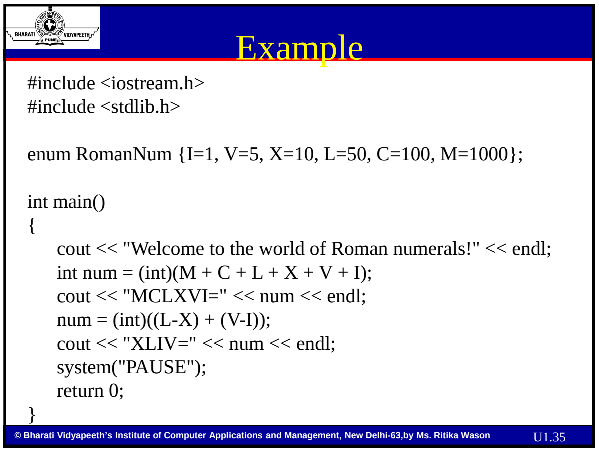
```
MonthType  month ;

for  (month = JAN ;  month <= DEC ;  month = MonthType (month + 1 ) )
{                                              // uses type cast to increment

                        .
                        .
                        .
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.33



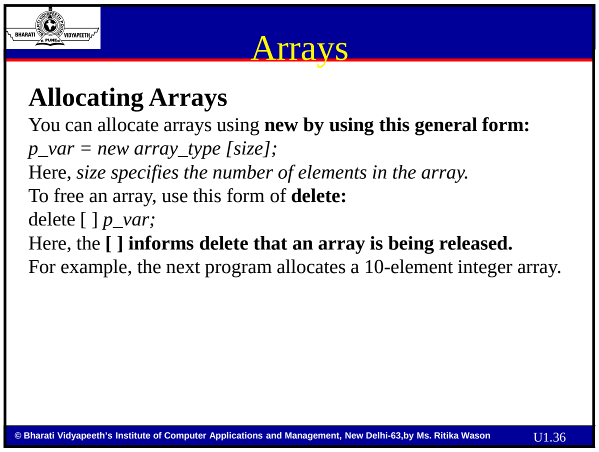
Simple arithmetic with enumeration constants

- Previously, we defined an enumeration of the symbols used in the Roman numeral system
- We will use this enumeration to illustrate arithmetic with enums, and see another example of type casting.
- Note: The insertion (<<) and extraction (>>) operators are not defined for enum-type variables
- To perform input or output operations on user-defined types, you must provide your own functions.



Example

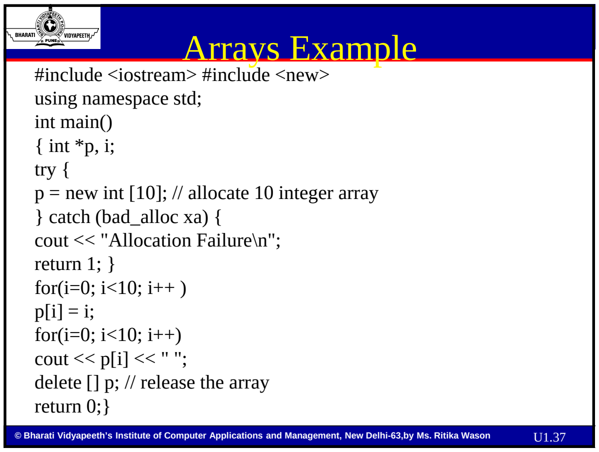
```
#include <iostream.h>
#include <stdlib.h>

enum RomanNum {I=1, V=5, X=10, L=50, C=100, M=1000};

int main()
{
    cout << "Welcome to the world of Roman numerals!" << endl;
    int num = (int)(M + C + L + X + V + I);
    cout << "MCLXVI=" << num << endl;
    num = (int)((L-X) + (V-I));
    cout << "XLIV=" << num << endl;
    system("PAUSE");
    return 0;
}
```



Arrays

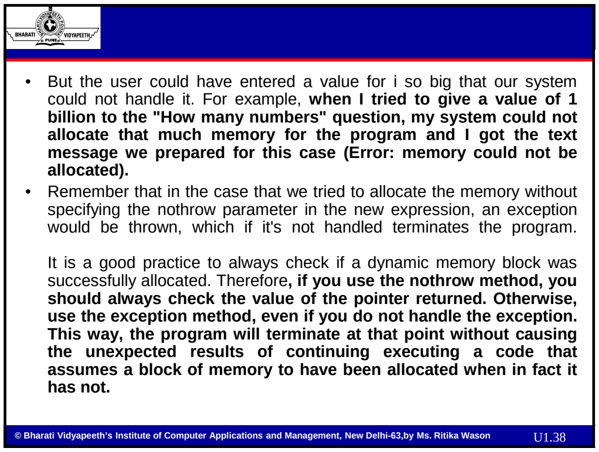
Allocating Arrays

You can allocate arrays using **new by using this general form:**
p_var = new array_type [size];
Here, *size specifies the number of elements in the array.*
To free an array, use this form of **delete:**
delete [ ] *p_var;*
Here, the **[ ] informs delete that an array is being released.**
For example, the next program allocates a 10-element integer array.



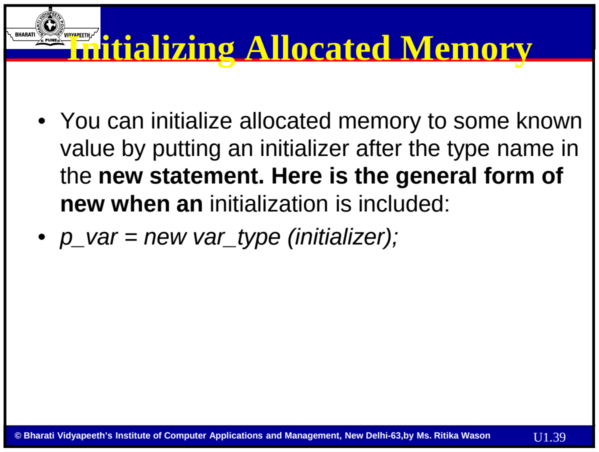
Arrays Example

```
#include <iostream> #include <new>
using namespace std;
int main()
{ int *p, i;
try {
p = new int [10]; // allocate 10 integer array
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1; }
for(i=0; i<10; i++ )
p[i] = i;
for(i=0; i<10; i++)
cout << p[i] << " ";
delete [] p; // release the array
return 0;}
```



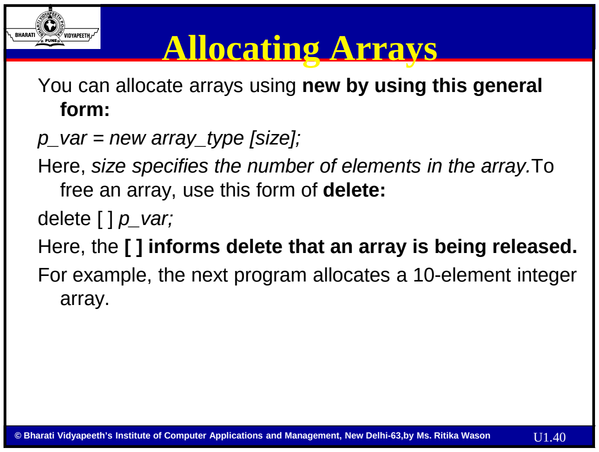
- But the user could have entered a value for i so big that our system could not handle it. For example, **when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated).**
- Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

  It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore**, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.**



Initializing Allocated Memory

- You can initialize allocated memory to some known value by putting an initializer after the type name in the **new statement. Here is the general form of new when an** initialization is included:
- *p_var = new var_type (initializer);*



Allocating Arrays

You can allocate arrays using **new by using this general form:**

p_var = new array_type [size];

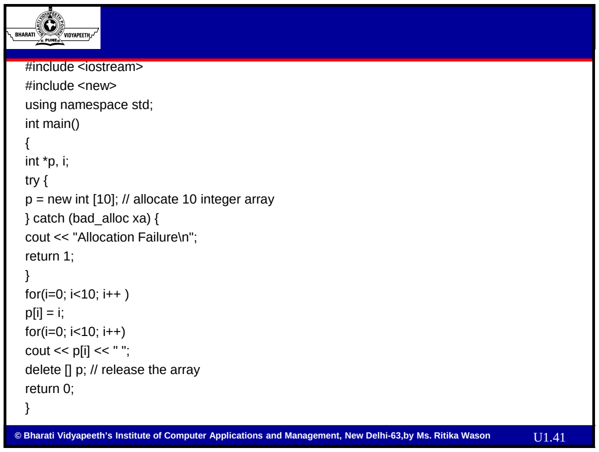
Here, *size specifies the number of elements in the array.*To free an array, use this form of **delete:**

delete [ ] *p_var;*

Here, the **[ ] informs delete that an array is being released.**

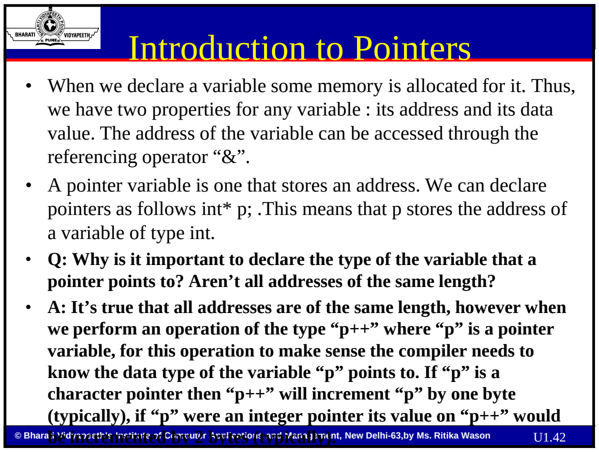
For example, the next program allocates a 10-element integer array.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.40



```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p, i;
try {
p = new int [10]; // allocate 10 integer array
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
for(i=0; i<10; i++ )
p[i] = i;
for(i=0; i<10; i++)
cout << p[i] << " ";
delete [] p; // release the array
return 0;
}
```

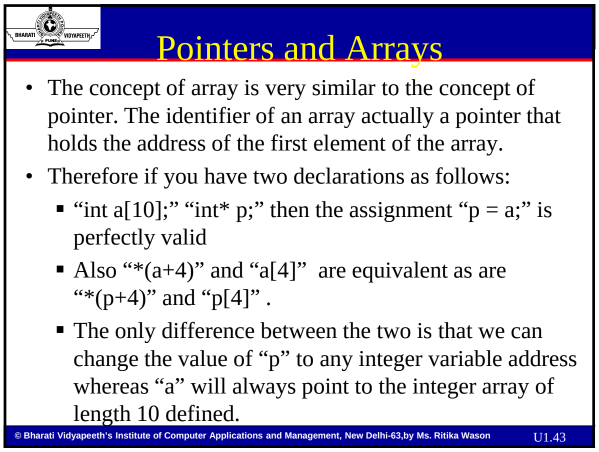
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.41



Introduction to Pointers

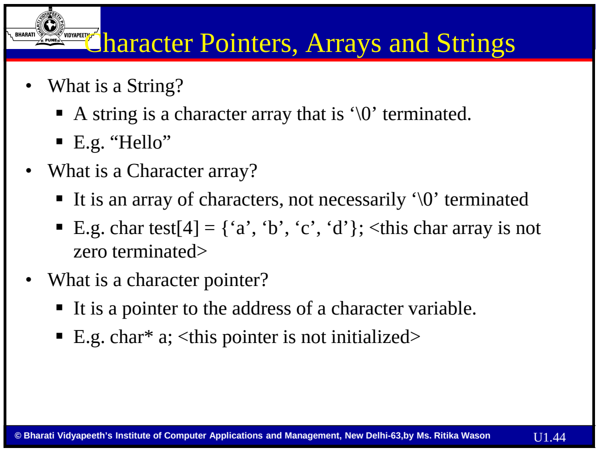
- When we declare a variable some memory is allocated for it. Thus, we have two properties for any variable : its address and its data value. The address of the variable can be accessed through the referencing operator "&".
- A pointer variable is one that stores an address. We can declare pointers as follows int* p; .This means that p stores the address of a variable of type int.
- **Q: Why is it important to declare the type of the variable that a pointer points to? Aren't all addresses of the same length?**
- **A: It's true that all addresses are of the same length, however when we perform an operation of the type "p++" where "p" is a pointer variable, for this operation to make sense the compiler needs to know the data type of the variable "p" points to. If "p" is a character pointer then "p++" will increment "p" by one byte (typically), if "p" were an integer pointer its value on "p++" would be incremented by 2 bytes (typically).**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.42



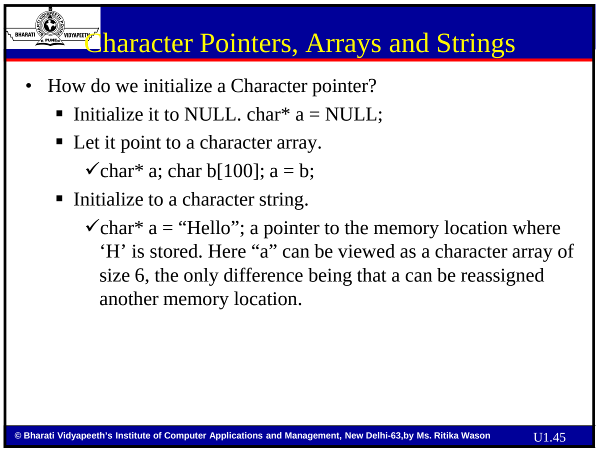
Pointers and Arrays

- The concept of array is very similar to the concept of pointer. The identifier of an array actually a pointer that holds the address of the first element of the array.
- Therefore if you have two declarations as follows:
  - “int a[10];” “int* p;” then the assignment “p = a;” is perfectly valid
  - Also “*(a+4)” and “a[4]” are equivalent as are “*(p+4)” and “p[4]” .
  - The only difference between the two is that we can change the value of “p” to any integer variable address whereas “a” will always point to the integer array of length 10 defined.



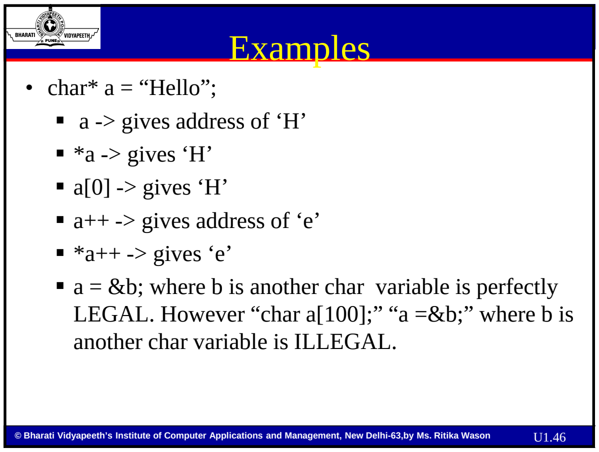
Character Pointers, Arrays and Strings

- What is a String?
  - A string is a character array that is ‘\0’ terminated.
  - E.g. “Hello”
- What is a Character array?
  - It is an array of characters, not necessarily ‘\0’ terminated
  - E.g. char test[4] = {‘a’, ‘b’, ‘c’, ‘d’}; <this char array is not zero terminated>
- What is a character pointer?
  - It is a pointer to the address of a character variable.
  - E.g. char* a; <this pointer is not initialized>



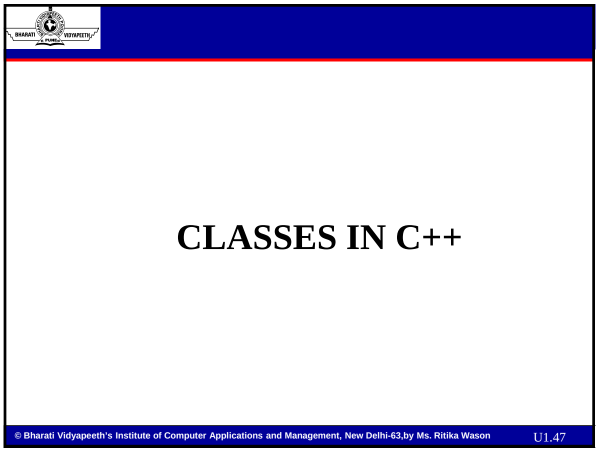
Character Pointers, Arrays and Strings

- How do we initialize a Character pointer?
  - Initialize it to NULL. char* a = NULL;
  - Let it point to a character array.
    - ✓char* a; char b[100]; a = b;
  - Initialize to a character string.
    - ✓char* a = “Hello”; a pointer to the memory location where ‘H’ is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

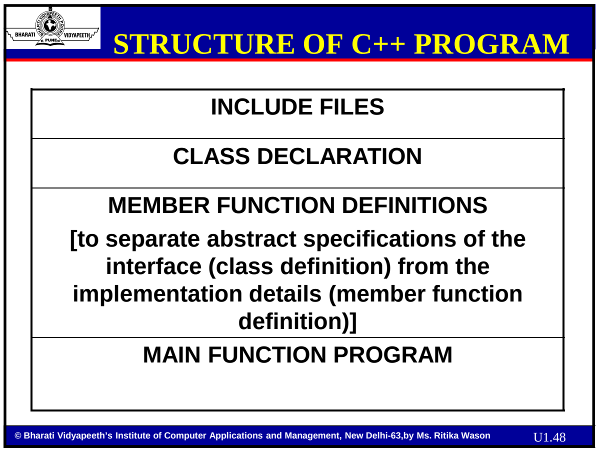


Examples

- char* a = “Hello”;
  - a -> gives address of ‘H’
  - *a -> gives ‘H’
  - a[0] -> gives ‘H’
  - a++ -> gives address of ‘e’
  - *a++ -> gives ‘e’
  - a = &b; where b is another char variable is perfectly LEGAL. However “char a[100];” “a =&b;” where b is another char variable is ILLEGAL.



CLASSES IN C++



STRUCTURE OF C++ PROGRAM

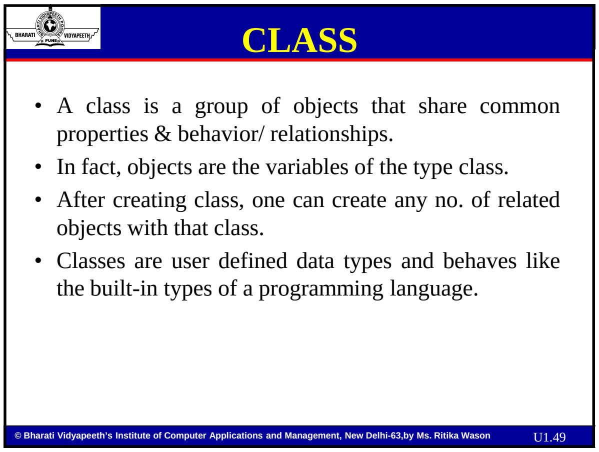
INCLUDE FILES

CLASS DECLARATION

MEMBER FUNCTION DEFINITIONS

[to separate abstract specifications of the interface (class definition) from the implementation details (member function definition)]

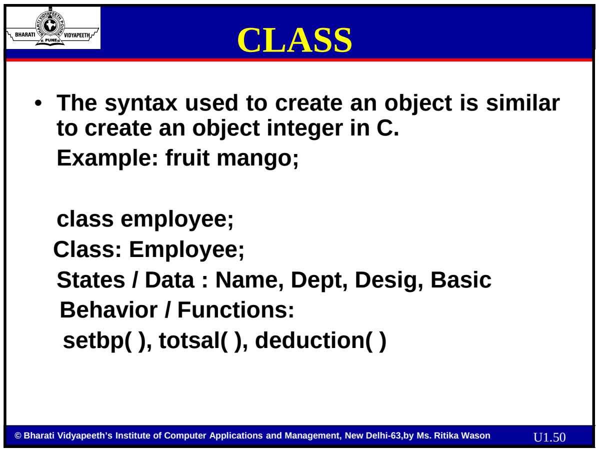
MAIN FUNCTION PROGRAM



CLASS

- A class is a group of objects that share common properties & behavior/ relationships.
- In fact, objects are the variables of the type class.
- After creating class, one can create any no. of related objects with that class.
- Classes are user defined data types and behaves like the built-in types of a programming language.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.49

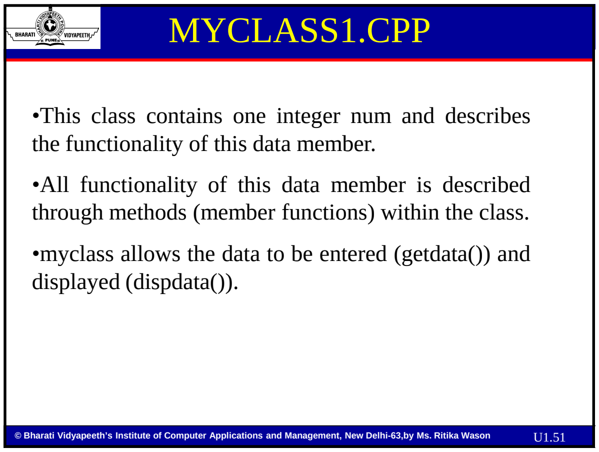


CLASS

- **The syntax used to create an object is similar to create an object integer in C.**
  Example: fruit mango;

  class employee;
  Class: Employee;
  States / Data : Name, Dept, Desig, Basic
  Behavior / Functions:
  setbp( ), totsal( ), deduction( )

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.50



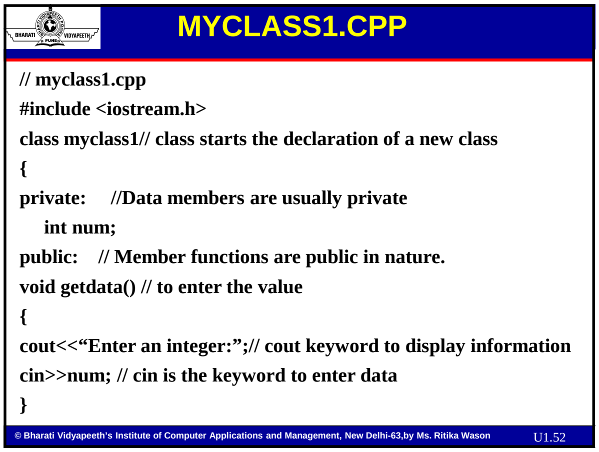
MYCLASS1.CPP

•This class contains one integer num and describes the functionality of this data member.

•All functionality of this data member is described through methods (member functions) within the class.

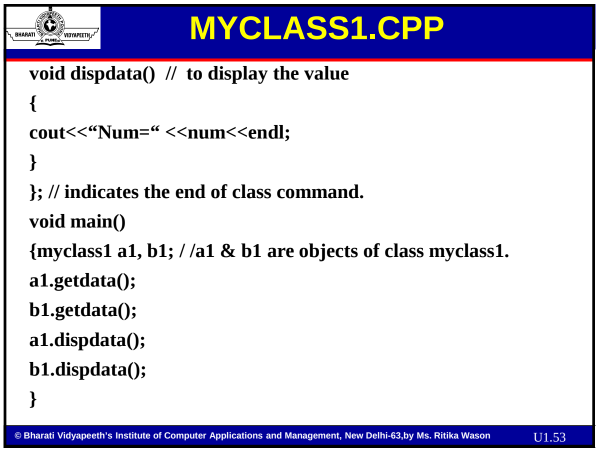
•myclass allows the data to be entered (getdata()) and displayed (dispdata()).

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.51



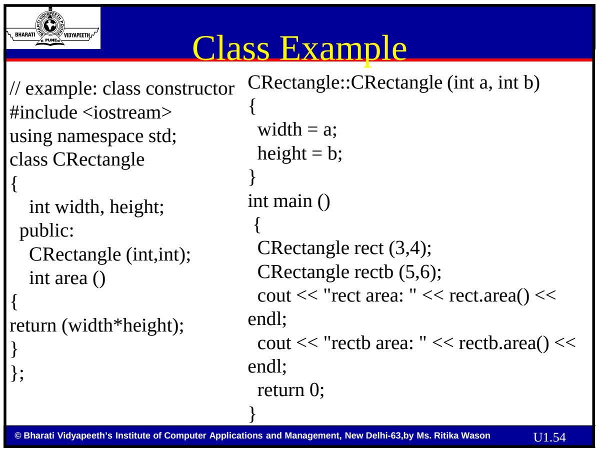
MYCLASS1.CPP

```
// myclass1.cpp
#include <iostream.h>
class myclass1// class starts the declaration of a new class
{
private:    //Data members are usually private
    int num;
public:   // Member functions are public in nature.
void getdata() // to enter the value
{
cout<<"Enter an integer:";// cout keyword to display information
cin>>num; // cin is the keyword to enter data
}
```



MYCLASS1.CPP

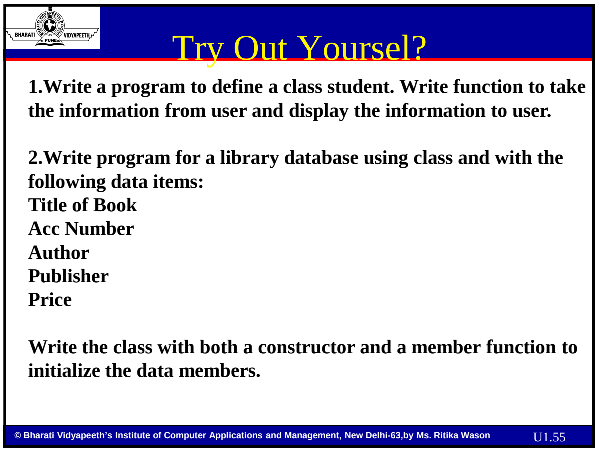
```
void dispdata()  //  to display the value
{
cout<<"Num=" <<num<<endl;
}
}; // indicates the end of class command.
void main()
{myclass1 a1, b1; / /a1 & b1 are objects of class myclass1.
a1.getdata();
b1.getdata();
a1.dispdata();
b1.dispdata();
}
```



Class Example

```
// example: class constructor
#include <iostream>
using namespace std;
class CRectangle
{
   int width, height;
 public:
   CRectangle (int,int);
   int area ()
{
return (width*height);
}
};
```

```
CRectangle::CRectangle (int a, int b)
{
  width = a;
  height = b;
}
int main ()
 {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() <<
endl;
  cout << "rectb area: " << rectb.area() <<
endl;
  return 0;
}
```

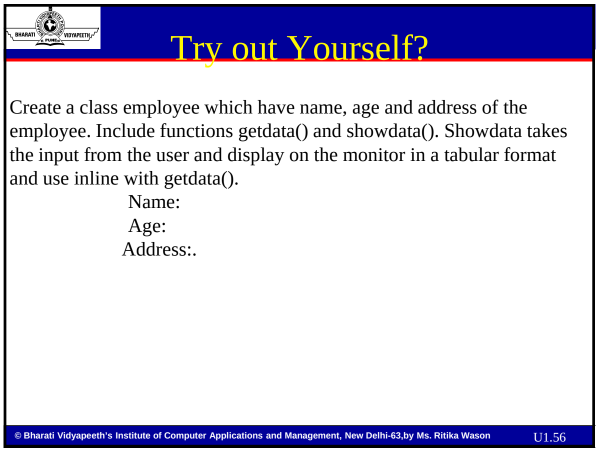


Try Out Yoursel?

1.Write a program to define a class student. Write function to take the information from user and display the information to user.

2.Write program for a library database using class and with the following data items:
Title of Book
Acc Number
Author
Publisher
Price

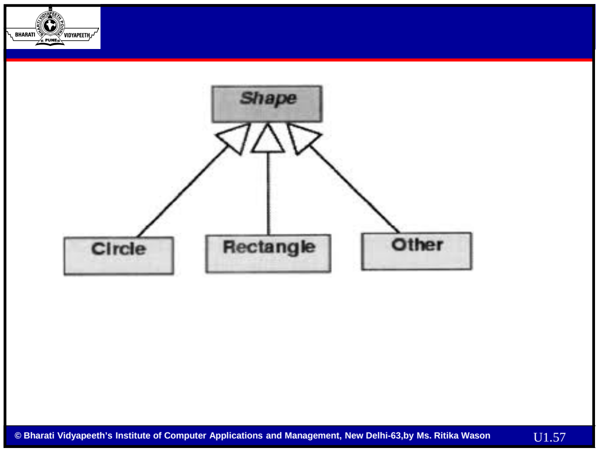
Write the class with both a constructor and a member function to initialize the data members.

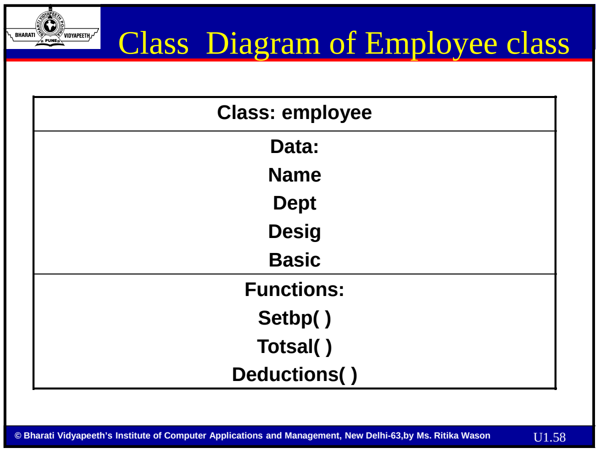


Try out Yourself?

Create a class employee which have name, age and address of the employee. Include functions getdata() and showdata(). Showdata takes the input from the user and display on the monitor in a tabular format and use inline with getdata().

Name:
Age:
Address:.



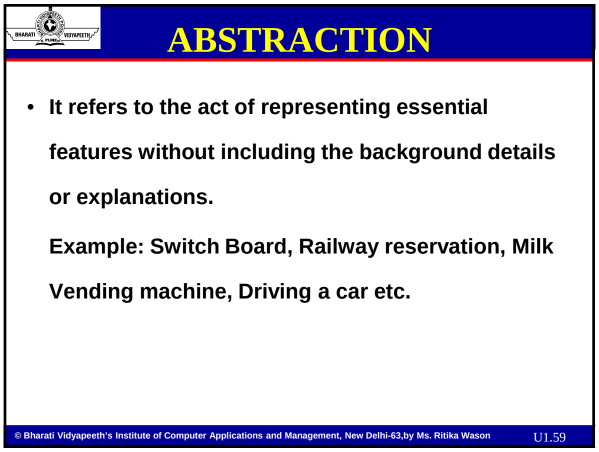


Class Diagram of Employee class

Class: employee
Data: **Name** **Dept** **Desig** **Basic**

Functions:
Setbp( )
Totsal( )
Deductions( )

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.58

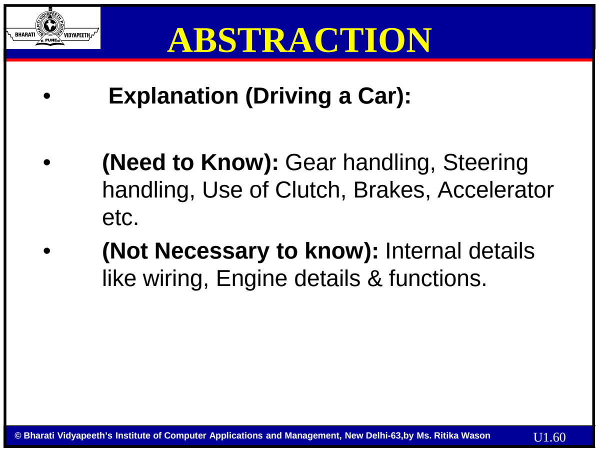


ABSTRACTION

- **It refers to the act of representing essential features without including the background details or explanations.**

  Example: Switch Board, Railway reservation, Milk Vending machine, Driving a car etc.

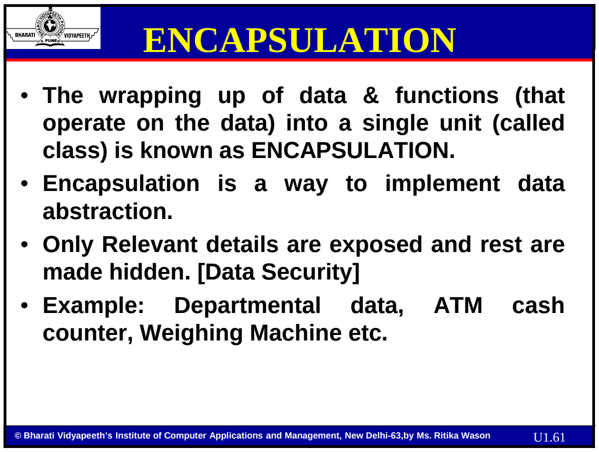
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.59



ABSTRACTION

- **Explanation (Driving a Car):**
- **(Need to Know):** Gear handling, Steering handling, Use of Clutch, Brakes, Accelerator etc.
- **(Not Necessary to know):** Internal details like wiring, Engine details & functions.

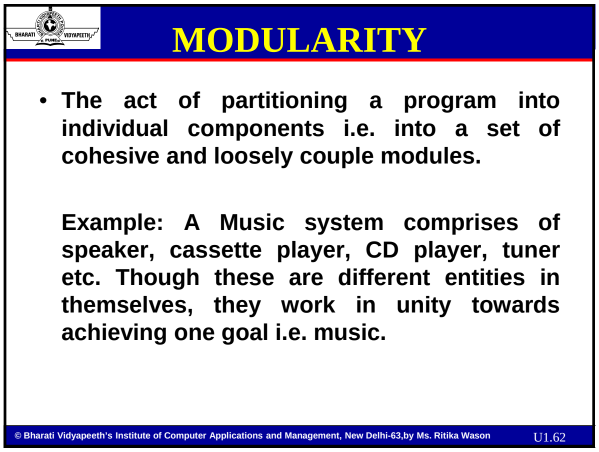
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.60



ENCAPSULATION

- The wrapping up of data & functions (that operate on the data) into a single unit (called class) is known as ENCAPSULATION.
- Encapsulation is a way to implement data abstraction.
- Only Relevant details are exposed and rest are made hidden. [Data Security]
- Example: Departmental data, ATM cash counter, Weighing Machine etc.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.61

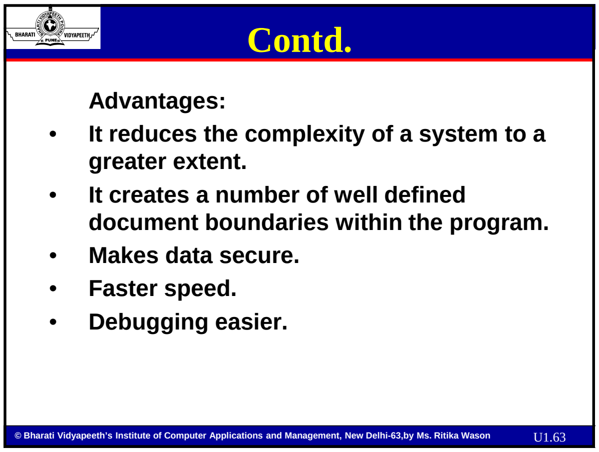


MODULARITY

- The act of partitioning a program into individual components i.e. into a set of cohesive and loosely couple modules.

  Example: A Music system comprises of speaker, cassette player, CD player, tuner etc. Though these are different entities in themselves, they work in unity towards achieving one goal i.e. music.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.62

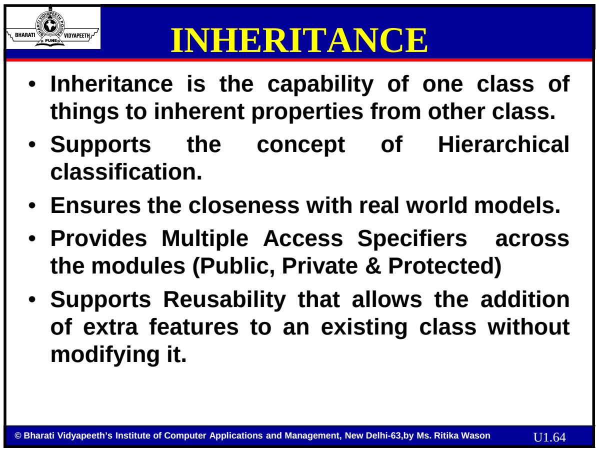


Contd.

Advantages:

- It reduces the complexity of a system to a greater extent.
- It creates a number of well defined document boundaries within the program.
- Makes data secure.
- Faster speed.
- Debugging easier.

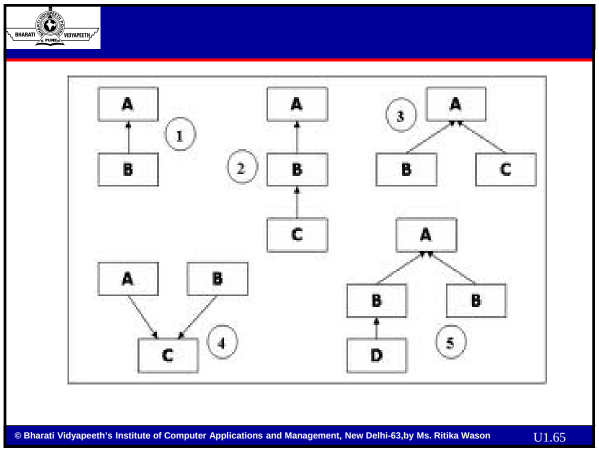
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.63



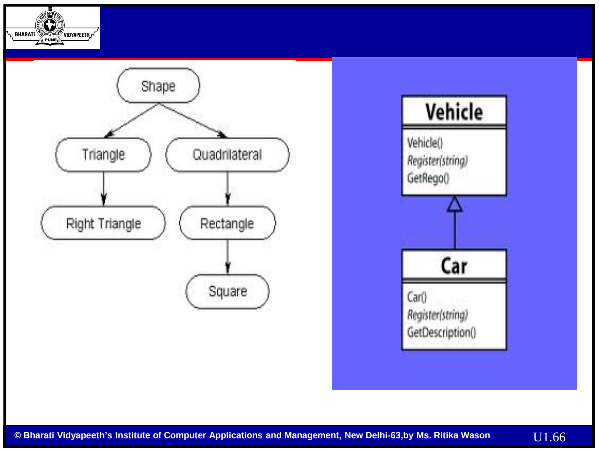
INHERITANCE

- Inheritance is the capability of one class of things to inherent properties from other class.
- Supports the concept of Hierarchical classification.
- Ensures the closeness with real world models.
- Provides Multiple Access Specifiers across the modules (Public, Private & Protected)
- Supports Reusability that allows the addition of extra features to an existing class without modifying it.

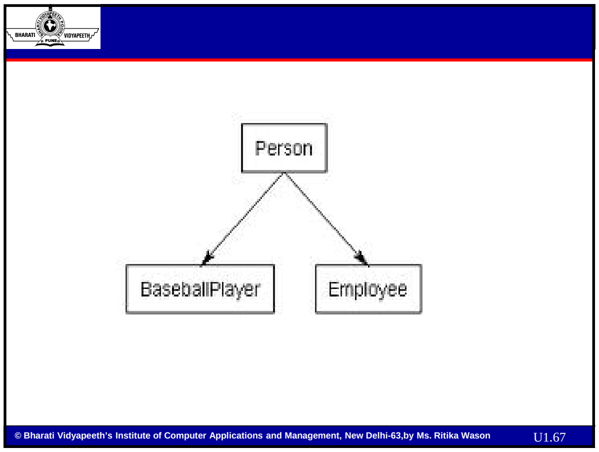
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.64



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.65



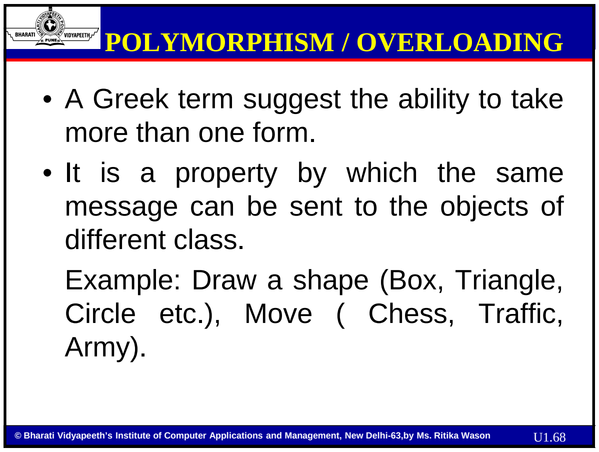
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.66



Person

BaseballPlayer

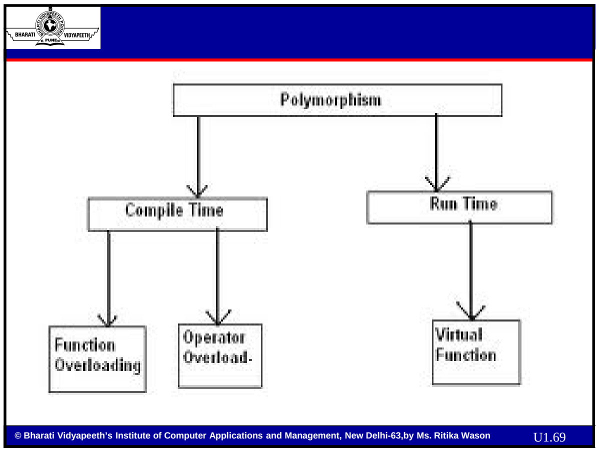
Employee



POLYMORPHISM / OVERLOADING

- A Greek term suggest the ability to take more than one form.
- It is a property by which the same message can be sent to the objects of different class.

  Example: Draw a shape (Box, Triangle, Circle etc.), Move (Chess, Traffic, Army).



Polymorphism

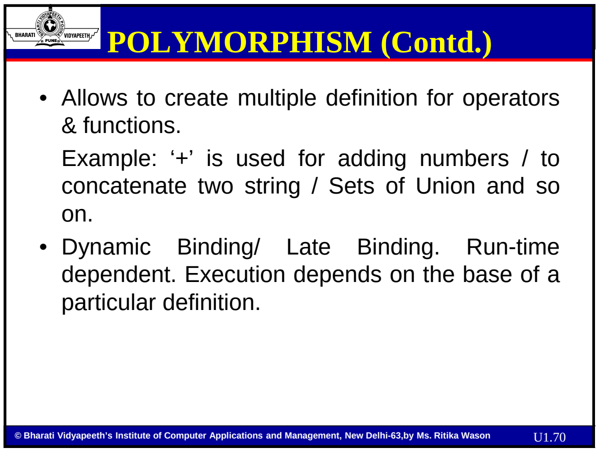
Compile Time

Run Time

Function Overloading

Operator Overload-

Virtual Function

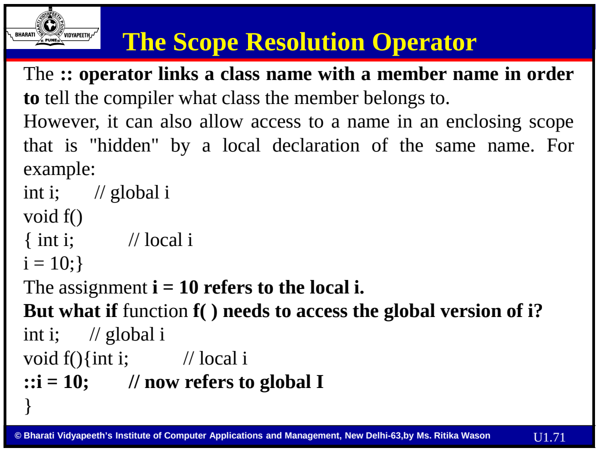


POLYMORPHISM (Contd.)

- Allows to create multiple definition for operators & functions.

  Example: ‘+’ is used for adding numbers / to concatenate two string / Sets of Union and so on.
- Dynamic Binding/ Late Binding. Run-time dependent. Execution depends on the base of a particular definition.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.70



The Scope Resolution Operator

The **:: operator links a class name with a member name in order to** tell the compiler what class the member belongs to.

However, it can also allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example:

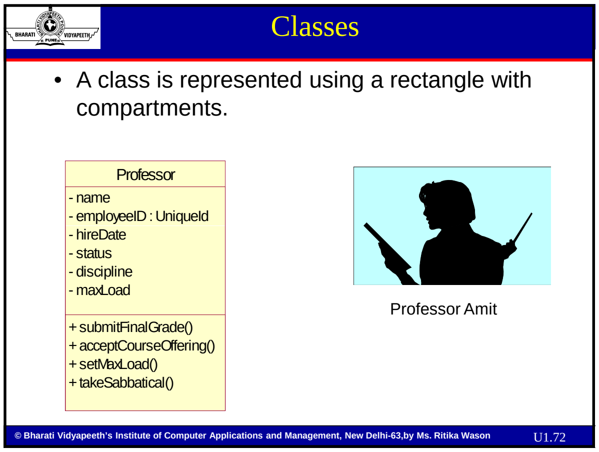
```
int i;      // global i
void f()
{ int i;        // local i
i = 10;}
```

The assignment **i = 10 refers to the local i.**

But what if function **f( ) needs to access the global version of i?**

```
int i;     // global i
void f(){int i;         // local i
::i = 10;      // now refers to global I
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.71



Classes

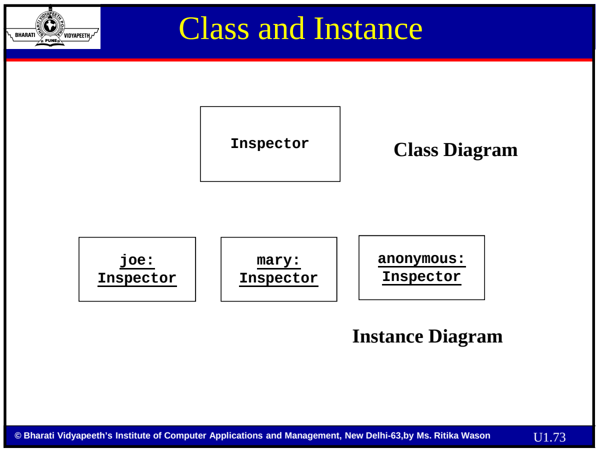
- A class is represented using a rectangle with compartments.

Professor
- name - employeeID : UniqueId - hireDate - status - discipline - maxLoad

+ submitFinalGrade()
+ acceptCourseOffering()
+ setMaxLoad()
+ takeSabbatical()

Professor Amit

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.72



Class and Instance

Inspector

Class Diagram

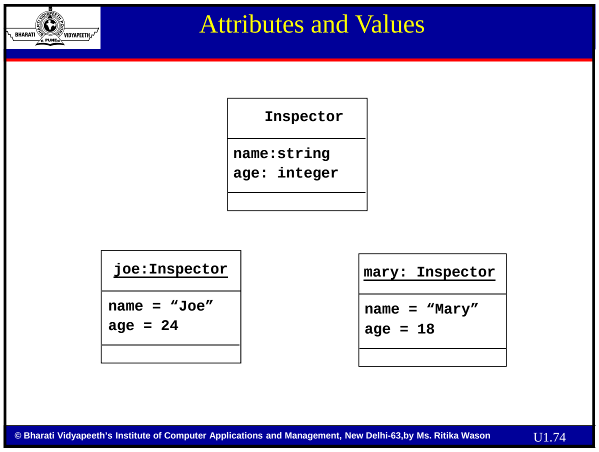
joe: Inspector

mary: Inspector

anonymous: Inspector

Instance Diagram

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.73



Attributes and Values

Inspector

name:string
age: integer

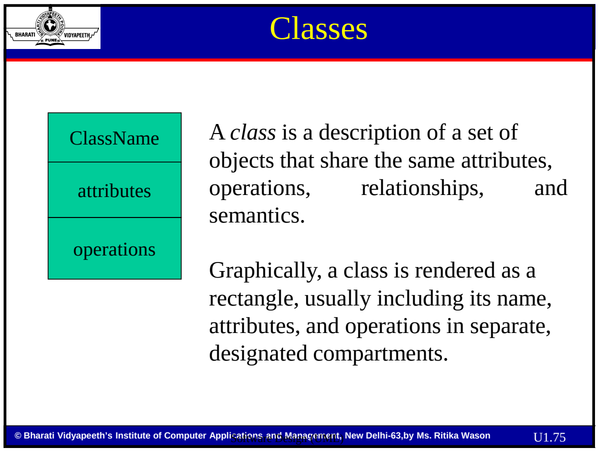
joe:Inspector

name = “Joe”
age = 24

mary: Inspector

name = “Mary”
age = 18

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.74



Classes

ClassName

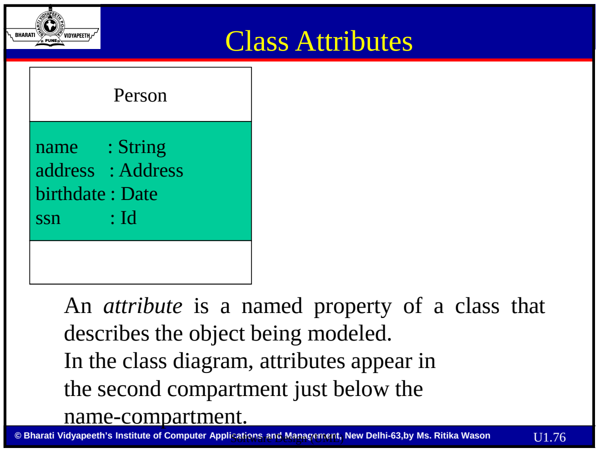
attributes

operations

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.75

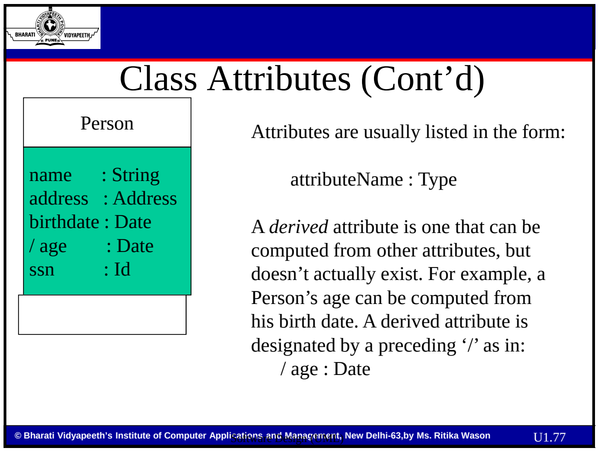


Class Attributes

Person

name : String
address : Address
birthdate : Date
ssn : Id

An *attribute* is a named property of a class that describes the object being modeled.
In the class diagram, attributes appear in the second compartment just below the name-compartment.



Class Attributes (Cont'd)

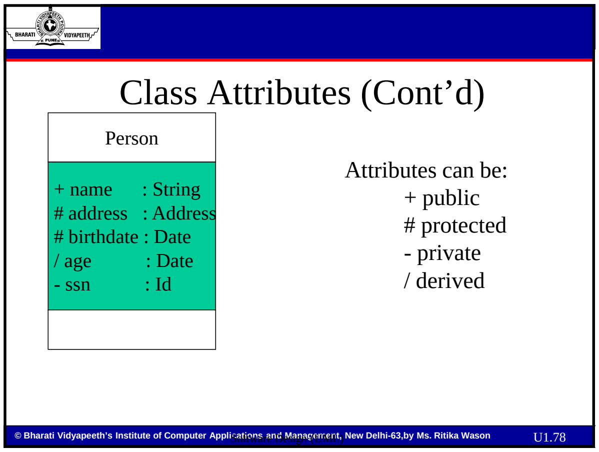
Person

name : String
address : Address
birthdate : Date
/ age : Date
ssn : Id

Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:
/ age : Date

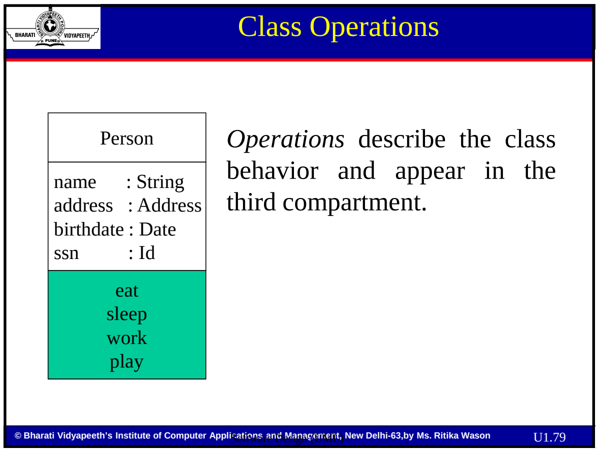


Class Attributes (Cont'd)

Person

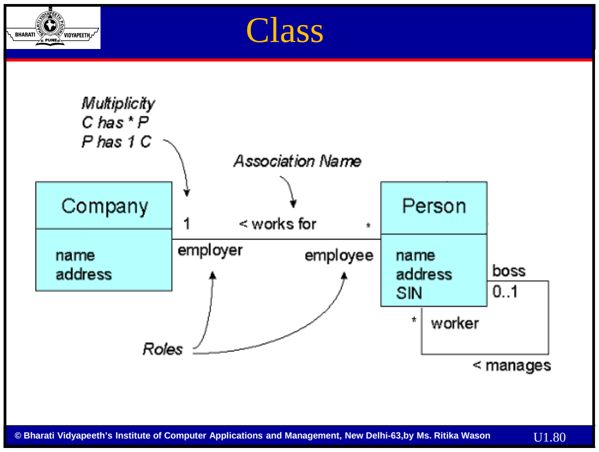
+ name : String
# address : Address
# birthdate : Date
/ age : Date
- ssn : Id

Attributes can be:
+ public
# protected
- private
/ derived

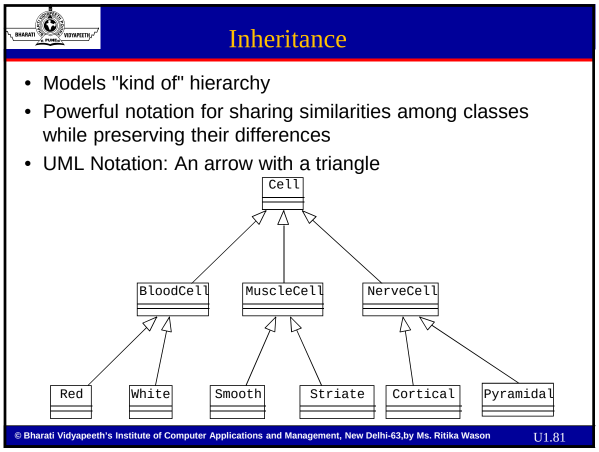


Class Operations

Operations describe the class behavior and appear in the third compartment.



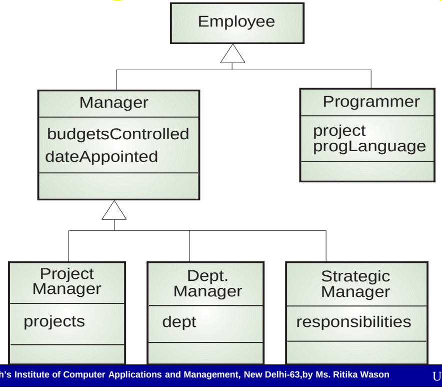
Class



Inheritance

- Models "kind of" hierarchy
- Powerful notation for sharing similarities among classes while preserving their differences
- UML Notation: An arrow with a triangle

A generalisation hierarchy



Employee

Manager

budgetsControlled
dateAppointed

Programmer

project
progLanguage

Project Manager

projects

Dept. Manager

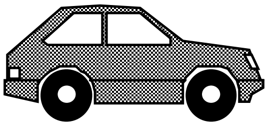
dept

Strategic Manager

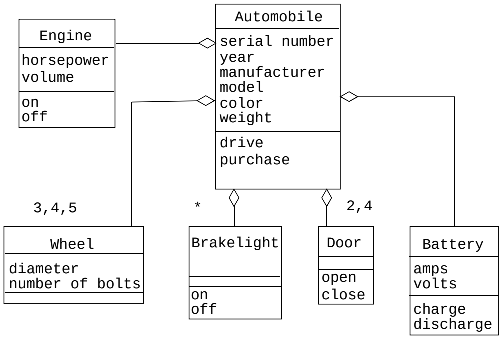
responsibilities

Aggregation

- Models "part of" hierarchy
- Useful for modeling the breakdown of a product into **its component** parts .
- UML notation: Like an association but with a small diamond indicating the assembly end of the relationship.



Aggregation



Automobile

serial number
year
manufacturer
model
color
weight

drive
purchase

Engine

horsepower
volume

on
off

3,4,5

Wheel

diameter
number of bolts

*

Brakelight

on
off

2,4

Door

open
close

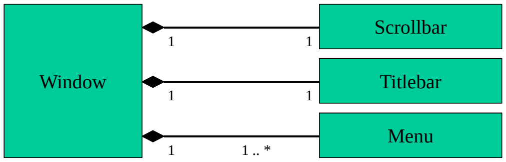
Battery

amps
volts

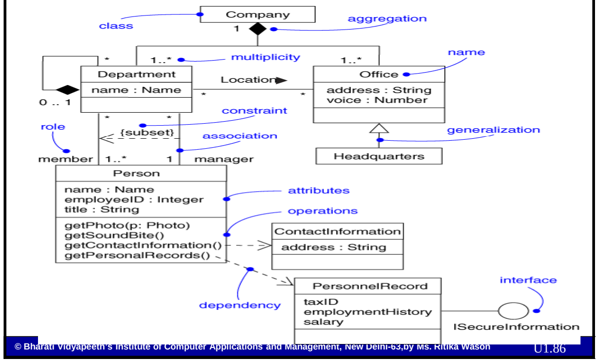
charge
discharge

Association Relationships

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



- Captures the vocabulary of a system



OBJECT

- An object can be an item, place, person or any other entity. All objects have the following characteristics:

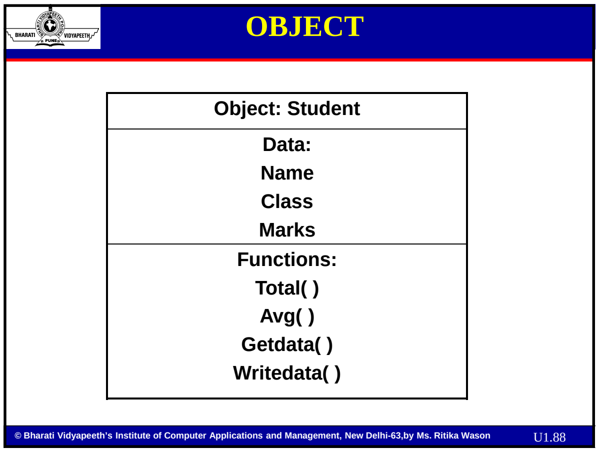
  Identity: The name associated with an object helps in identifying the object. Example: Play-Ground, Multiplexes, Wall-clock, Class-Room, Court-Room

  State: An object can be in many state.

  Example: TV can be in the following states: On State, Off State, Out of order state.

  Behavior: What the object does or what is it capable of doing?

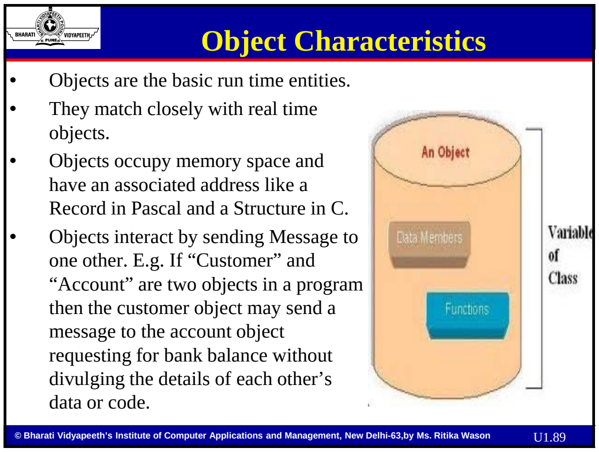
  Example: A person can sit, stand, read, sleep, walk & talk etc.



OBJECT

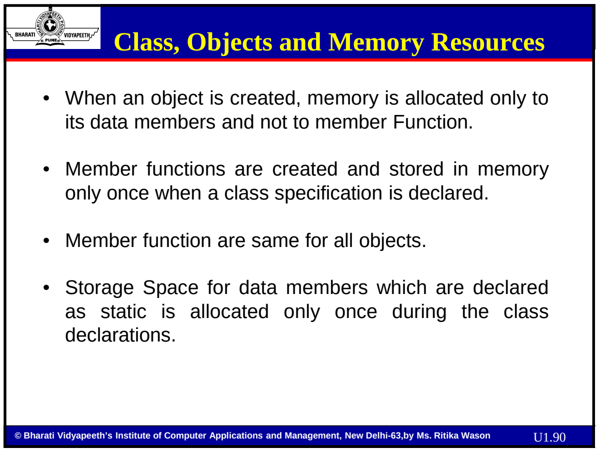
Object: Student
Data: Name Class Marks

Functions:
Total( )
Avg( )
Getdata( )
Writedata( )



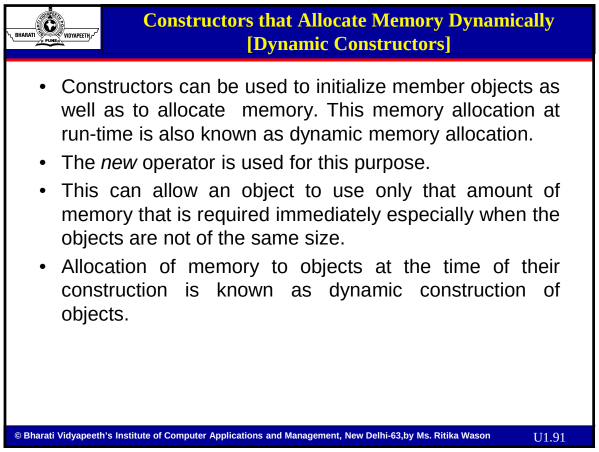
Object Characteristics

- Objects are the basic run time entities.
- They match closely with real time objects.
- Objects occupy memory space and have an associated address like a Record in Pascal and a Structure in C.
- Objects interact by sending Message to one other. E.g. If “Customer” and “Account” are two objects in a program then the customer object may send a message to the account object requesting for bank balance without divulging the details of each other’s data or code.



Class, Objects and Memory Resources

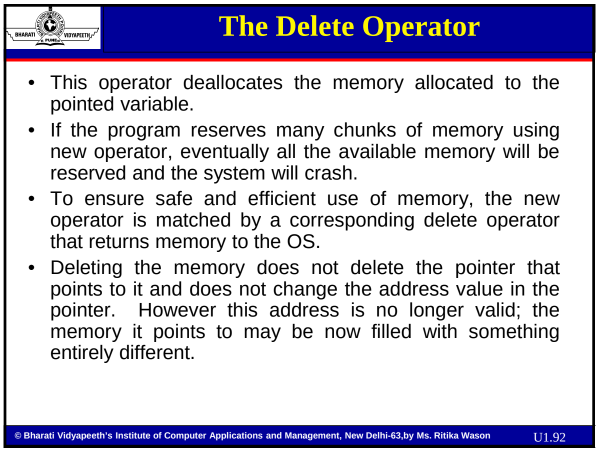
- When an object is created, memory is allocated only to its data members and not to member Function.
- Member functions are created and stored in memory only once when a class specification is declared.
- Member function are same for all objects.
- Storage Space for data members which are declared as static is allocated only once during the class declarations.



Constructors that Allocate Memory Dynamically [Dynamic Constructors]

- Constructors can be used to initialize member objects as well as to allocate memory. This memory allocation at run-time is also known as dynamic memory allocation.
- The *new* operator is used for this purpose.
- This can allow an object to use only that amount of memory that is required immediately especially when the objects are not of the same size.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.

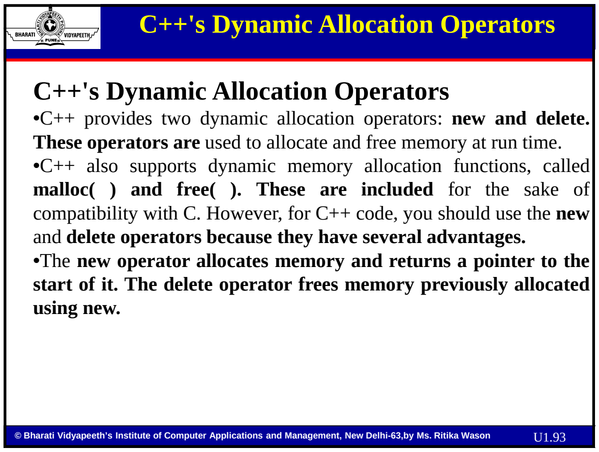
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.91



The Delete Operator

- This operator deallocates the memory allocated to the pointed variable.
- If the program reserves many chunks of memory using new operator, eventually all the available memory will be reserved and the system will crash.
- To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the OS.
- Deleting the memory does not delete the pointer that points to it and does not change the address value in the pointer. However this address is no longer valid; the memory it points to may be now filled with something entirely different.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.92



C++'s Dynamic Allocation Operators

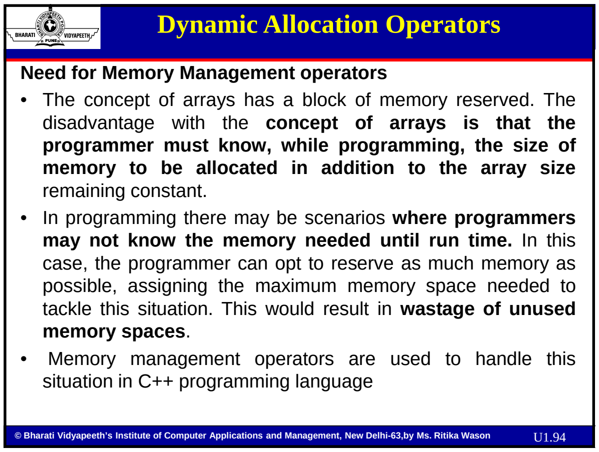
C++'s Dynamic Allocation Operators

•C++ provides two dynamic allocation operators: **new and delete. These operators are** used to allocate and free memory at run time.

•C++ also supports dynamic memory allocation functions, called **malloc( ) and free( ). These are included** for the sake of compatibility with C. However, for C++ code, you should use the **new** and **delete operators because they have several advantages.**

•The **new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.93

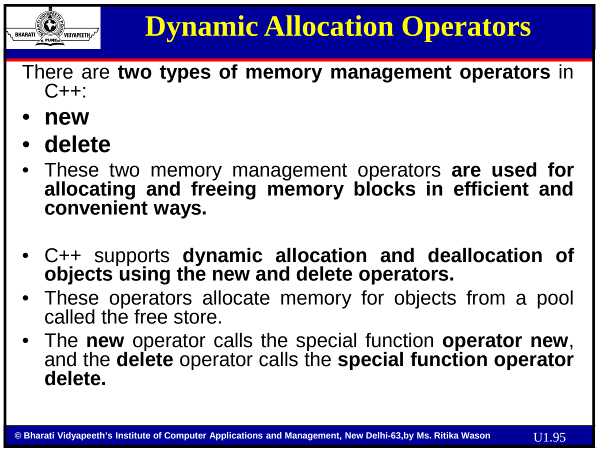


Dynamic Allocation Operators

Need for Memory Management operators

- The concept of arrays has a block of memory reserved. The disadvantage with the **concept of arrays is that the programmer must know, while programming, the size of memory to be allocated in addition to the array size** remaining constant.
- In programming there may be scenarios **where programmers may not know the memory needed until run time.** In this case, the programmer can opt to reserve as much memory as possible, assigning the maximum memory space needed to tackle this situation. This would result in **wastage of unused memory spaces**.
- Memory management operators are used to handle this situation in C++ programming language

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.94

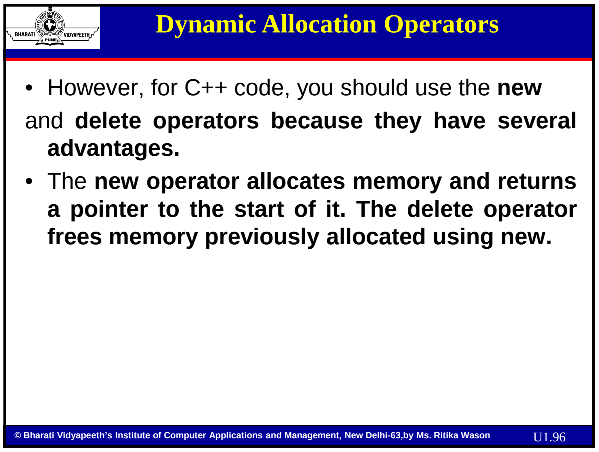


Dynamic Allocation Operators

There are **two types of memory management operators** in C++:

- **new**
- **delete**
- These two memory management operators **are used for allocating and freeing memory blocks in efficient and convenient ways.**
- C++ supports **dynamic allocation and deallocation of objects using the new and delete operators.**
- These operators allocate memory for objects from a pool called the free store.
- The **new** operator calls the special function **operator new**, and the **delete** operator calls the **special function operator delete.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.95



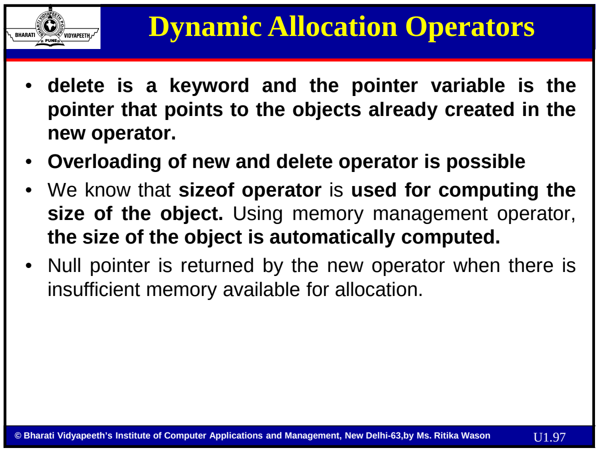
Dynamic Allocation Operators

- However, for C++ code, you should use the **new**

and **delete operators because they have several advantages.**

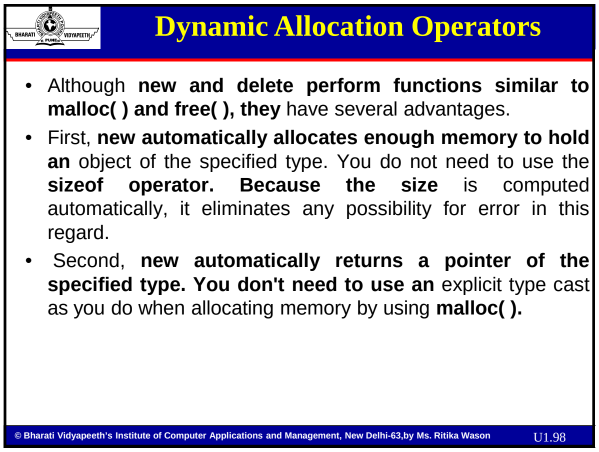
- The **new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.96



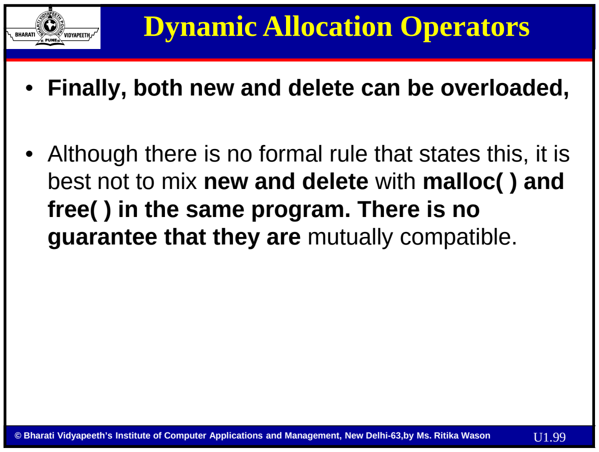
Dynamic Allocation Operators

- **delete is a keyword and the pointer variable is the pointer that points to the objects already created in the new operator.**
- **Overloading of new and delete operator is possible**
- We know that **sizeof operator** is **used for computing the size of the object.** Using memory management operator, **the size of the object is automatically computed.**
- Null pointer is returned by the new operator when there is insufficient memory available for allocation.



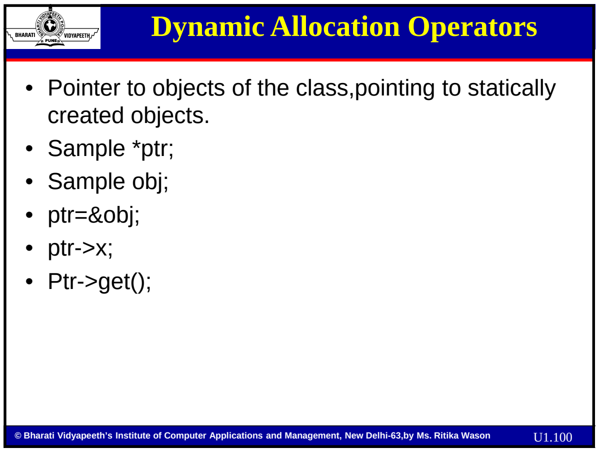
Dynamic Allocation Operators

- Although **new and delete perform functions similar to malloc( ) and free( ), they** have several advantages.
- First, **new automatically allocates enough memory to hold an** object of the specified type. You do not need to use the **sizeof operator. Because the size** is computed automatically, it eliminates any possibility for error in this regard.
- Second, **new automatically returns a pointer of the specified type. You don't need to use an** explicit type cast as you do when allocating memory by using **malloc( ).**



Dynamic Allocation Operators

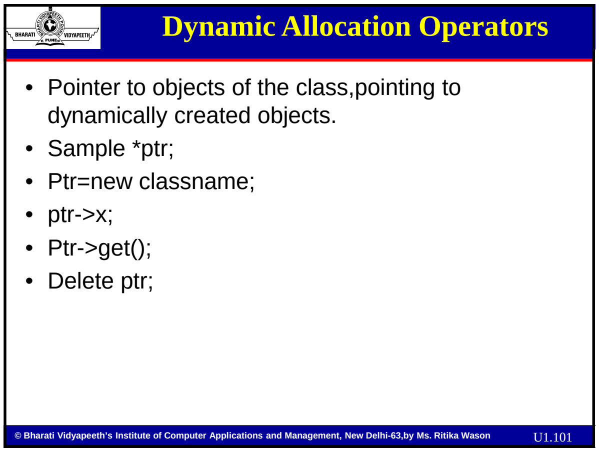
- **Finally, both new and delete can be overloaded,**
- Although there is no formal rule that states this, it is best not to mix **new and delete** with **malloc( ) and free( ) in the same program. There is no guarantee that they are** mutually compatible.



Dynamic Allocation Operators

- Pointer to objects of the class,pointing to statically created objects.
- Sample *ptr;
- Sample obj;
- ptr=&obj;
- ptr->x;
- Ptr->get();

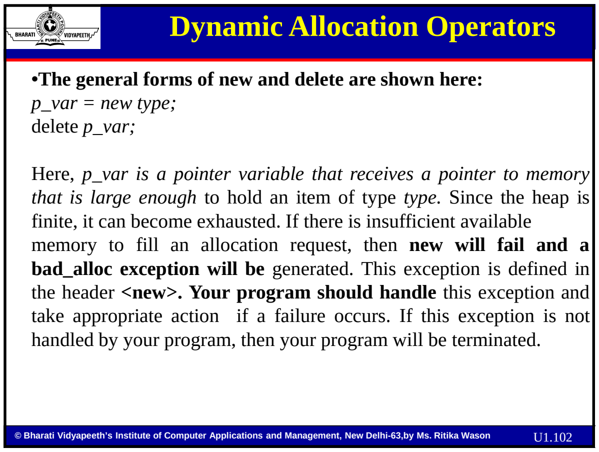
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.100



Dynamic Allocation Operators

- Pointer to objects of the class,pointing to dynamically created objects.
- Sample *ptr;
- Ptr=new classname;
- ptr->x;
- Ptr->get();
- Delete ptr;

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.101

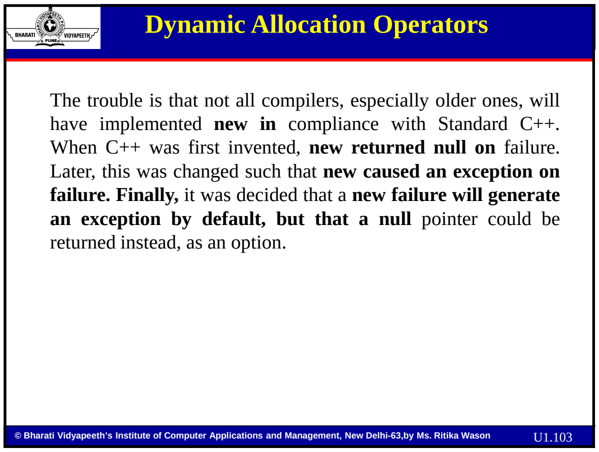


Dynamic Allocation Operators

•The general forms of new and delete are shown here:
p_var = new type;
delete *p_var;*

Here, *p_var is a pointer variable that receives a pointer to memory that is large enough* to hold an item of type *type.* Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new will fail and a bad_alloc exception will be** generated. This exception is defined in the header **<new>. Your program should handle** this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

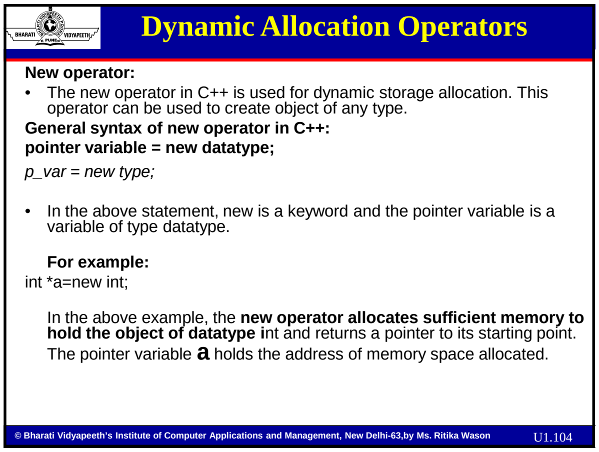
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.102



Dynamic Allocation Operators

The trouble is that not all compilers, especially older ones, will have implemented **new in** compliance with Standard C++. When C++ was first invented, **new returned null on** failure. Later, this was changed such that **new caused an exception on failure. Finally,** it was decided that a **new failure will generate an exception by default, but that a null** pointer could be returned instead, as an option.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.103



Dynamic Allocation Operators

New operator:

- The new operator in C++ is used for dynamic storage allocation. This operator can be used to create object of any type.

General syntax of new operator in C++:

pointer variable = new datatype;

p_var = new type;

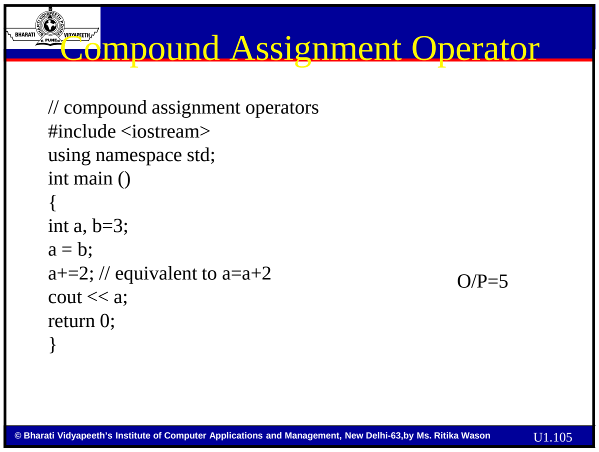
- In the above statement, new is a keyword and the pointer variable is a variable of type datatype.

For example:

int *a=new int;

In the above example, the **new operator allocates sufficient memory to hold the object of datatype i**nt and returns a pointer to its starting point. The pointer variable **a** holds the address of memory space allocated.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.104

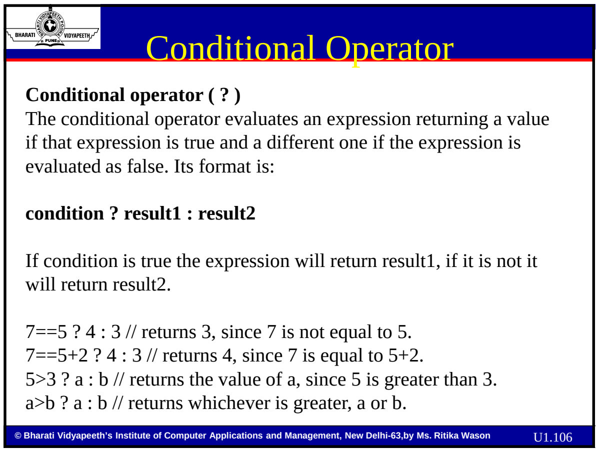


Compound Assignment Operator

```
// compound assignment operators
#include <iostream>
using namespace std;
int main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
```

O/P=5

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.105



Conditional Operator

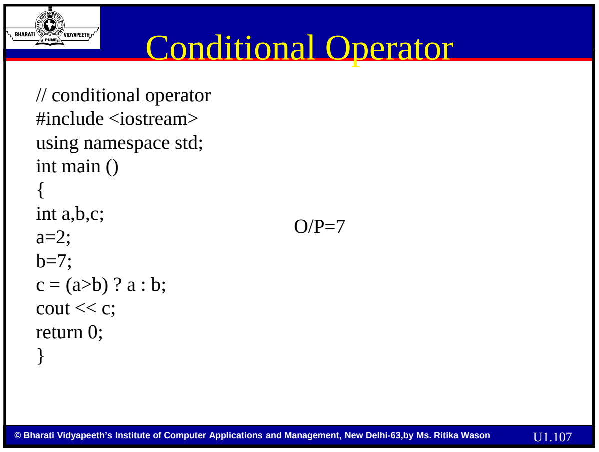
Conditional operator (?)
The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.106

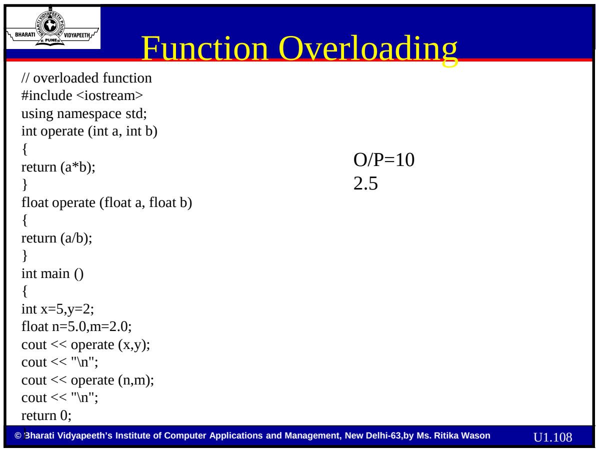


Conditional Operator

```
// conditional operator
#include <iostream>
using namespace std;
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
```

O/P=7

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.107

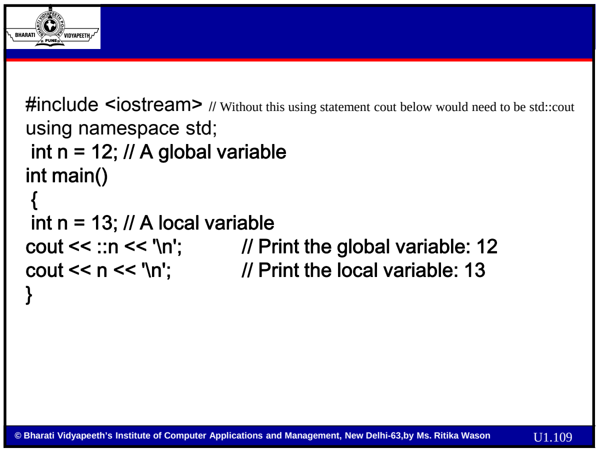


Function Overloading

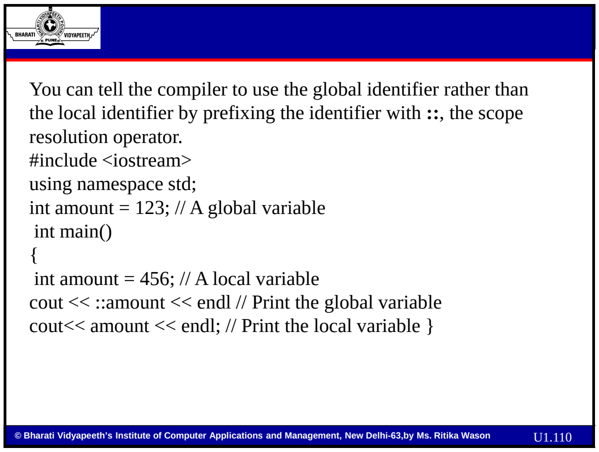
```
// overloaded function
#include <iostream>
using namespace std;
int operate (int a, int b)
{
return (a*b);
}
float operate (float a, float b)
{
return (a/b);
}
int main ()
{
int x=5,y=2;
float n=5.0,m=2.0;
cout << operate (x,y);
cout << "\n";
cout << operate (n,m);
cout << "\n";
return 0;
```

O/P=10
2.5

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.108

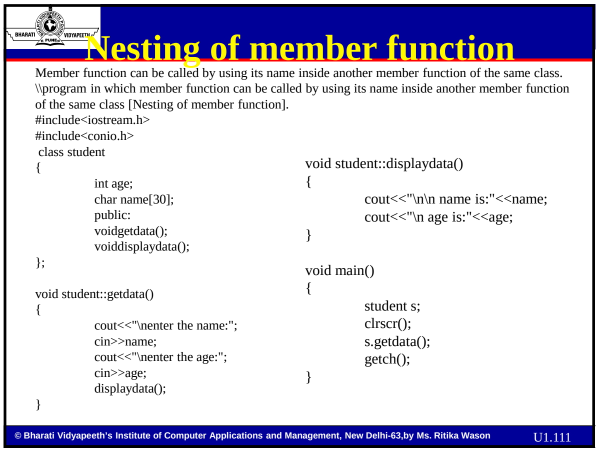


```
#include <iostream> // Without this using statement cout below would need to be std::cout
using namespace std;
 int n = 12; // A global variable
int main()
 {
 int n = 13; // A local variable
cout << ::n << '\n';          // Print the global variable: 12
cout << n << '\n';            // Print the local variable: 13
}
```



You can tell the compiler to use the global identifier rather than the local identifier by prefixing the identifier with **::**, the scope resolution operator.

```
#include <iostream>
using namespace std;
int amount = 123; // A global variable
 int main()
{
 int amount = 456; // A local variable
cout << ::amount << endl // Print the global variable
cout<< amount << endl; // Print the local variable }
```



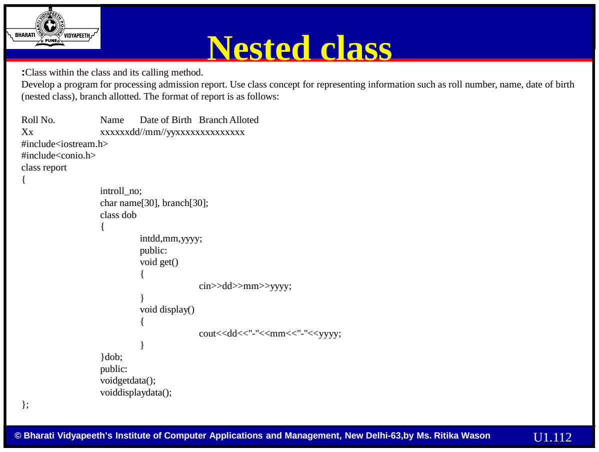
Nesting of member function

Member function can be called by using its name inside another member function of the same class.

\\program in which member function can be called by using its name inside another member function of the same class [Nesting of member function].

```
#include<iostream.h>
#include<conio.h>
 class student
{
        int age;
        char name[30];
        public:
        voidgetdata();
        voiddisplaydata();
};

void student::getdata()
{
        cout<<"\nenter the name:";
        cin>>name;
        cout<<"\nenter the age:";
        cin>>age;
        displaydata();
}
```

```
void student::displaydata()
{
        cout<<"\n\n name is:"<<name;
        cout<<"\n age is:"<<age;
}

void main()
{
        student s;
        clrscr();
        s.getdata();
        getch();
}
```



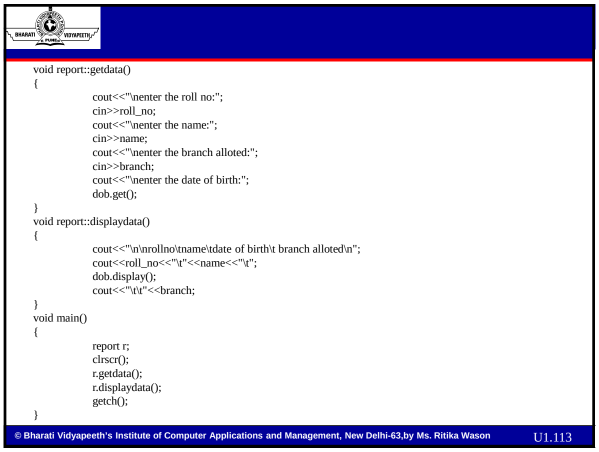
Nested class

:Class within the class and its calling method.

Develop a program for processing admission report. Use class concept for representing information such as roll number, name, date of birth (nested class), branch allotted. The format of report is as follows:

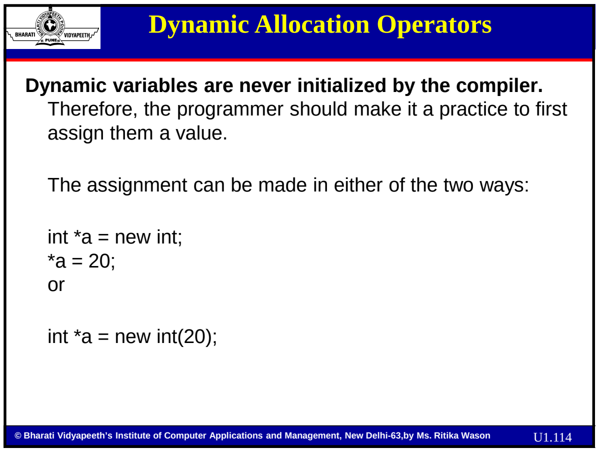
```
Roll No.            Name      Date of Birth   Branch Alloted
Xx                  xxxxxxdd//mm//yyxxxxxxxxxxxxxxx
#include<iostream.h>
#include<conio.h>
class report
{
                    introll_no;
                    char name[30], branch[30];
                    class dob
                    {
                              intdd,mm,yyyy;
                              public:
                              void get()
                              {
                                             cin>>dd>>mm>>yyyy;
                              }
                              void display()
                              {
                                             cout<<dd<<"-"<<mm<<"-"<<yyyy;
                              }
                    }dob;
                    public:
                    voidgetdata();
                    voiddisplaydata();
};
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.112



```
void report::getdata()
{
               cout<<"\nenter the roll no:";
               cin>>roll_no;
               cout<<"\nenter the name:";
               cin>>name;
               cout<<"\nenter the branch alloted:";
               cin>>branch;
               cout<<"\nenter the date of birth:";
               dob.get();
}
void report::displaydata()
{
               cout<<"\n\nrollno\tname\tdate of birth\t branch alloted\n";
               cout<<roll_no<<"\t"<<name<<"\t";
               dob.display();
               cout<<"\t\t"<<branch;
}
void main()
{
               report r;
               clrscr();
               r.getdata();
               r.displaydata();
               getch();
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.113



Dynamic Allocation Operators

Dynamic variables are never initialized by the compiler.

Therefore, the programmer should make it a practice to first assign them a value.

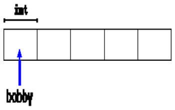
The assignment can be made in either of the two ways:

```
int *a = new int;
*a = 20;
or

int *a = new int(20);
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.114

Dynamic Allocation Operators

```
int * bobby;
bobby = new int [5];
```



•In this case, the **system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby.** Therefore, now, bobby points to a valid block of memory with space for five elements of type int.

•The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression *bobby. Both are equivalent as has been explained in the section about pointers.

•The second element can be accessed either with bobby[1] or *(bobby+1) and so on... The dynamic memory requested by our program is allocated by the system **from the memory heap**. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

delete operator:

The delete operator in C++ is used **for releasing memory space when the object is no longer needed.** Once a new operator is used, it is efficient to use the corresponding delete operator for release of memory.

delete pointer variable;

delete *p_var;*

```
#include <iostream.h>
    void main()
    {
    //Allocates using new operator memory space in memory for
    storing a integer datatype
    int *a= new a;
    *a=100;
    cout << " The Output is:a="<<*a;

    //Memory Released using delete operator
    delete a;
}
```

The output of the above program is

```
    The Output is:a=100
```

In the above program, the statement:

int *a= new a;

Holds memory space in memory for storing a integer datatype. The statement:

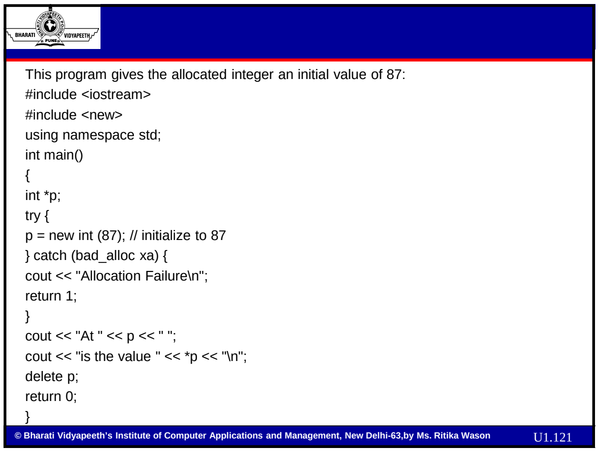
*a=100

- This denotes that the value present in address location pointed by the pointer **variable a** is 100 and this value of a is printed in the output statement giving the output shown in the example above.
- The memory allocated by the new operator for storing the integer variable pointed by a is released using the delete operator as:

delete a;

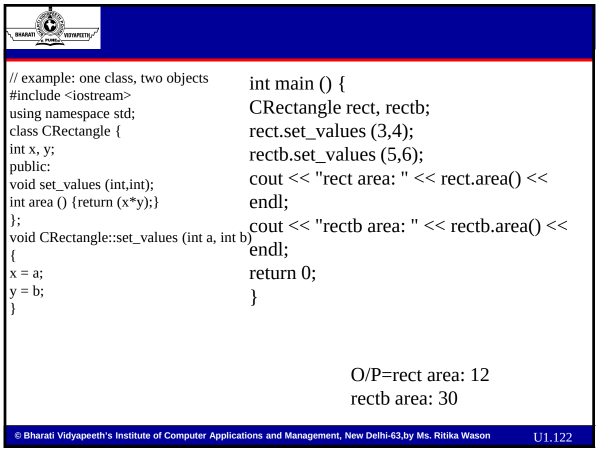
- Since the **heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request,** then **new will fail and a bad_alloc exception will be** generated.
- This exception is defined in the header **<new>. Your program should handle** this exception and take appropriate action if a failure occurs.

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int; // allocate space for an int
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
*p = 100;
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```



This program gives the allocated integer an initial value of 87:

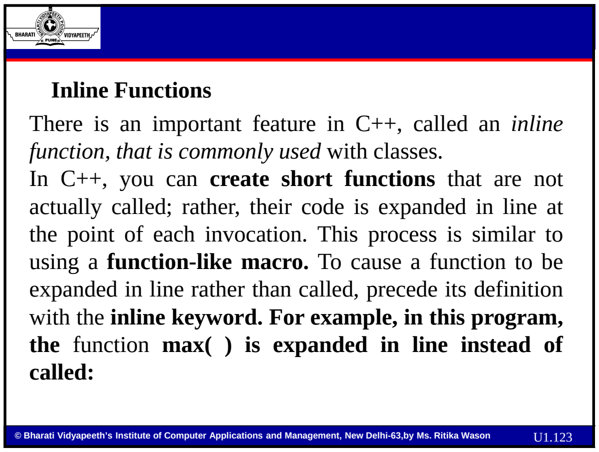
```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int (87); // initialize to 87
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```



```
// example: one class, two objects
#include <iostream>
using namespace std;
class CRectangle {
int x, y;
public:
void set_values (int,int);
int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b)
{
x = a;
y = b;
}
```

```
int main () {
CRectangle rect, rectb;
rect.set_values (3,4);
rectb.set_values (5,6);
cout << "rect area: " << rect.area() <<
endl;
cout << "rectb area: " << rectb.area() <<
endl;
return 0;
}
```

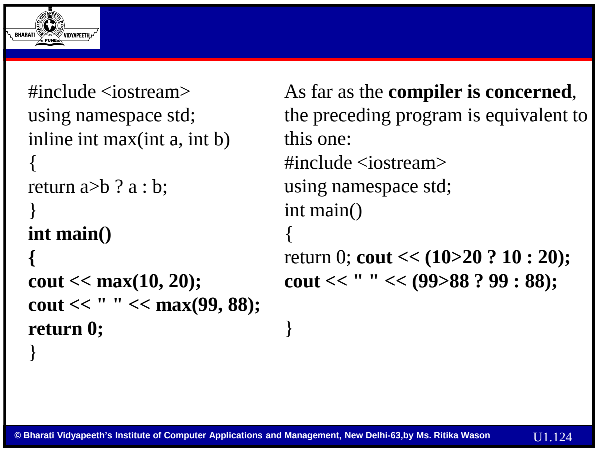
O/P=rect area: 12
rectb area: 30



Inline Functions

There is an important feature in C++, called an *inline function, that is commonly used* with classes.

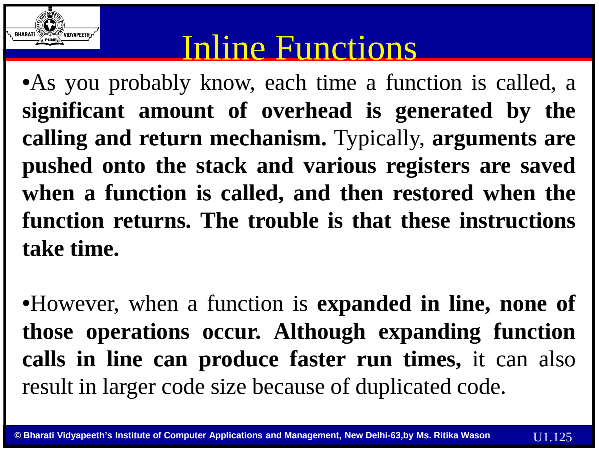
In C++, you can **create short functions** that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a **function-like macro.** To cause a function to be expanded in line rather than called, precede its definition with the **inline keyword. For example, in this program, the** function **max( ) is expanded in line instead of called:**



```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the **compiler is concerned**, the preceding program is equivalent to this one:

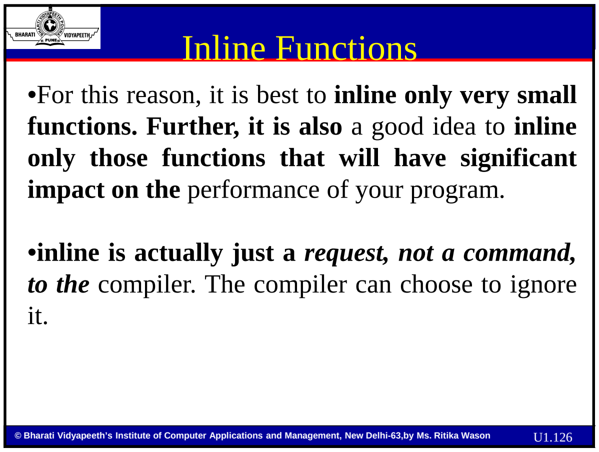
```
#include <iostream>
using namespace std;
int main()
{
return 0; cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);

}
```



Inline Functions

•As you probably know, each time a function is called, a **significant amount of overhead is generated by the calling and return mechanism.** Typically, **arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time.**

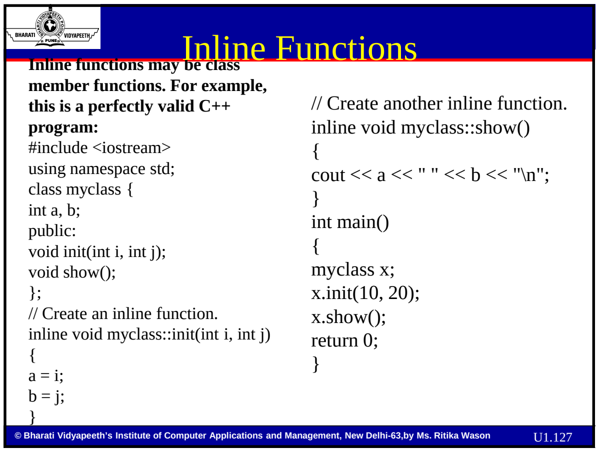
•However, when a function is **expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times,** it can also result in larger code size because of duplicated code.



Inline Functions

•For this reason, it is best to **inline only very small functions. Further, it is also** a good idea to **inline only those functions that will have significant impact on the** performance of your program.

•**inline is actually just a *request, not a command, to the*** compiler. The compiler can choose to ignore it.

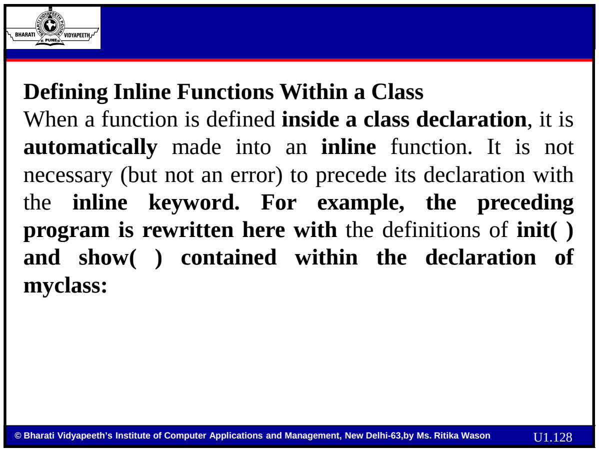


Inline Functions

Inline functions may be class member functions. For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
void init(int i, int j);
void show();
};
// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
// Create another inline function.
inline void myclass::show()
{
cout << a << " " << b << "\n";
}
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

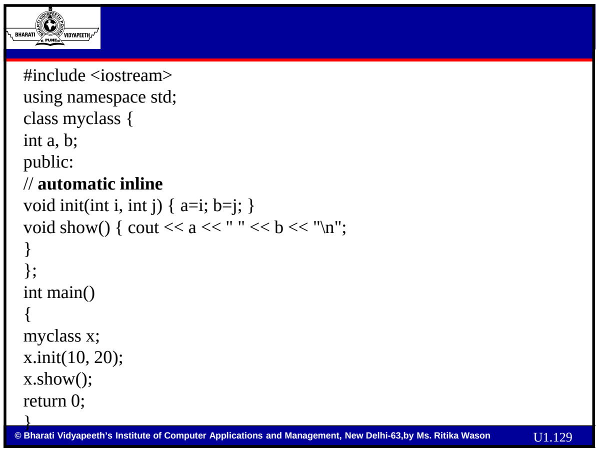
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.127



Defining Inline Functions Within a Class

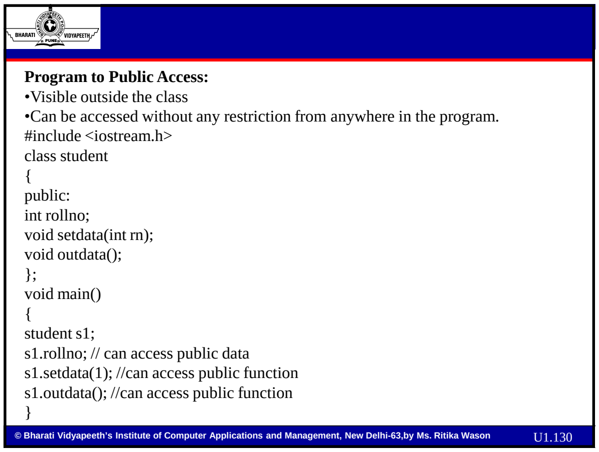
When a function is defined **inside a class declaration**, it is **automatically** made into an **inline** function. It is not necessary (but not an error) to precede its declaration with the **inline keyword. For example, the preceding program is rewritten here with** the definitions of **init( ) and show( ) contained within the declaration of myclass:**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.128



```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// automatic inline
void init(int i, int j) { a=i; b=j; }
void show() { cout << a << " " << b << "\n";
}
};
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

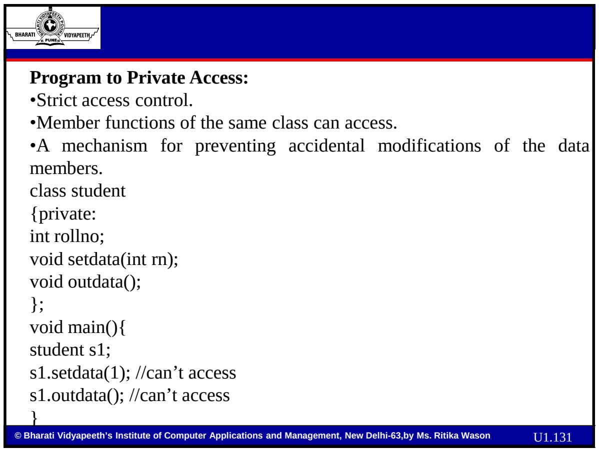
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.129



Program to Public Access:

- Visible outside the class
- Can be accessed without any restriction from anywhere in the program.

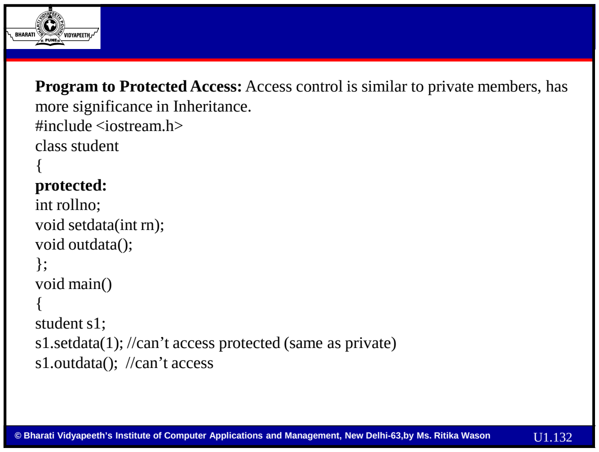
```
#include <iostream.h>
class student
{
public:
int rollno;
void setdata(int rn);
void outdata();
};
void main()
{
student s1;
s1.rollno; // can access public data
s1.setdata(1); //can access public function
s1.outdata(); //can access public function
}
```



Program to Private Access:

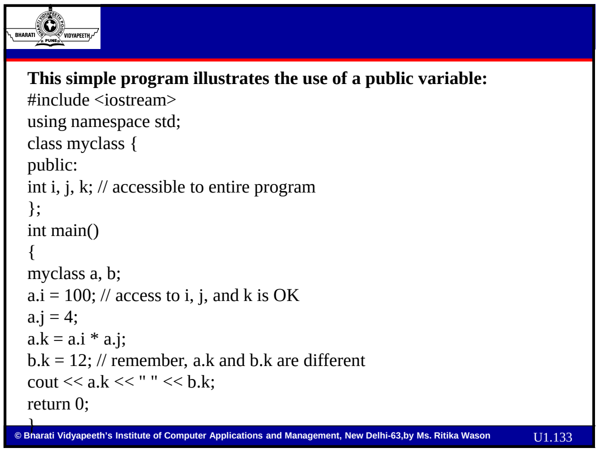
- Strict access control.
- Member functions of the same class can access.
- A mechanism for preventing accidental modifications of the data members.

```
class student
{private:
int rollno;
void setdata(int rn);
void outdata();
};
void main(){
student s1;
s1.setdata(1); //can’t access
s1.outdata(); //can’t access
}
```



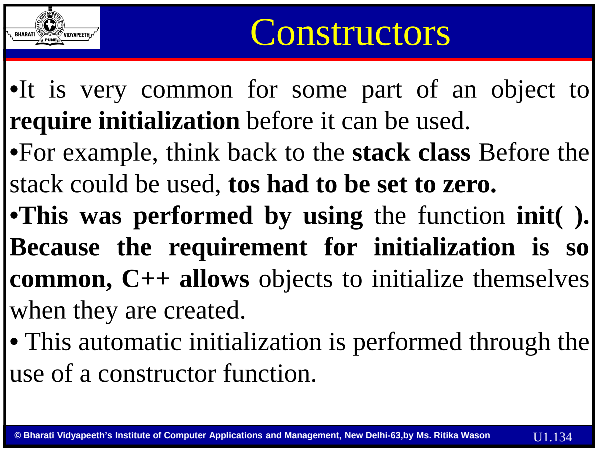
Program to Protected Access: Access control is similar to private members, has more significance in Inheritance.

```
#include <iostream.h>
class student
{
protected:
int rollno;
void setdata(int rn);
void outdata();
};
void main()
{
student s1;
s1.setdata(1); //can’t access protected (same as private)
s1.outdata();  //can’t access
```



This simple program illustrates the use of a public variable:

```
#include <iostream>
using namespace std;
class myclass {
public:
int i, j, k; // accessible to entire program
};
int main()
{
myclass a, b;
a.i = 100; // access to i, j, and k is OK
a.j = 4;
a.k = a.i * a.j;
b.k = 12; // remember, a.k and b.k are different
cout << a.k << " " << b.k;
return 0;
}
```



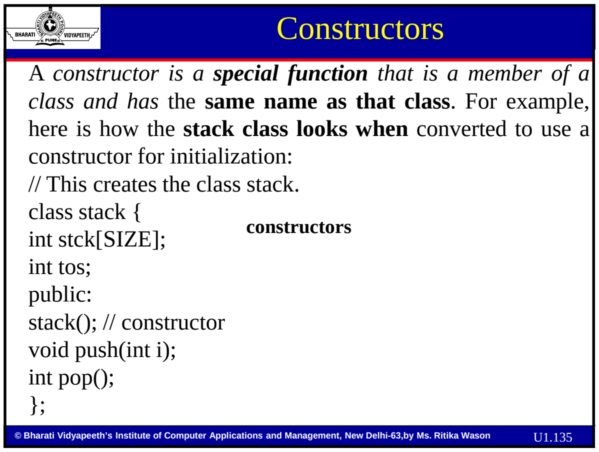
Constructors

•It is very common for some part of an object to **require initialization** before it can be used.

•For example, think back to the **stack class** Before the stack could be used, **tos had to be set to zero.**

•**This was performed by using** the function **init( ). Because the requirement for initialization is so common, C++ allows** objects to initialize themselves when they are created.

• This automatic initialization is performed through the use of a constructor function.

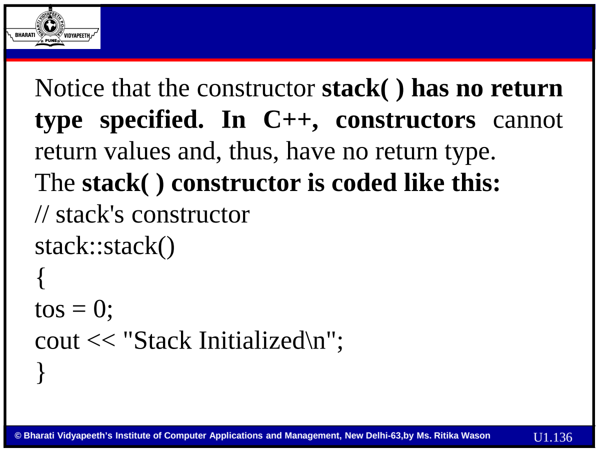


Constructors

A *constructor is a* ***special function*** *that is a member of a class and has* the **same name as that class**. For example, here is how the **stack class looks when** converted to use a constructor for initialization:

```
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
void push(int i);
int pop();
};
```

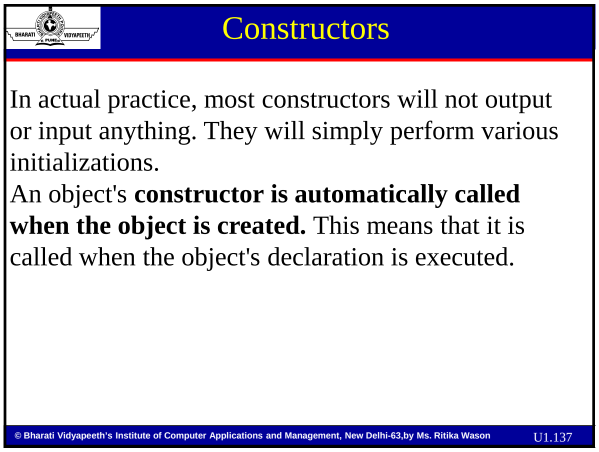
constructors



Notice that the constructor **stack( ) has no return type specified. In C++, constructors** cannot return values and, thus, have no return type.

The **stack( ) constructor is coded like this:**

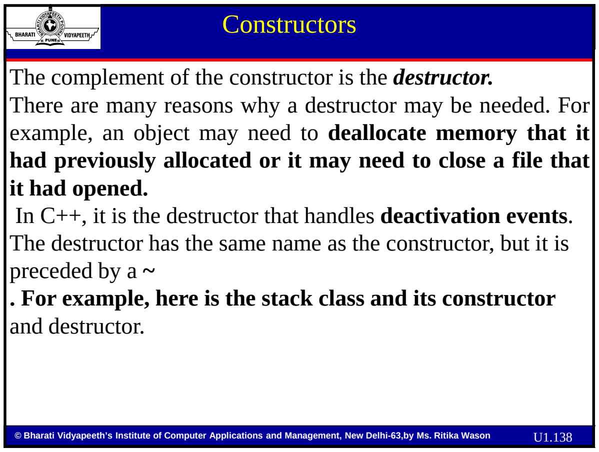
```
// stack's constructor
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
```



Constructors

In actual practice, most constructors will not output or input anything. They will simply perform various initializations.

An object's **constructor is automatically called when the object is created.** This means that it is called when the object's declaration is executed.



Constructors

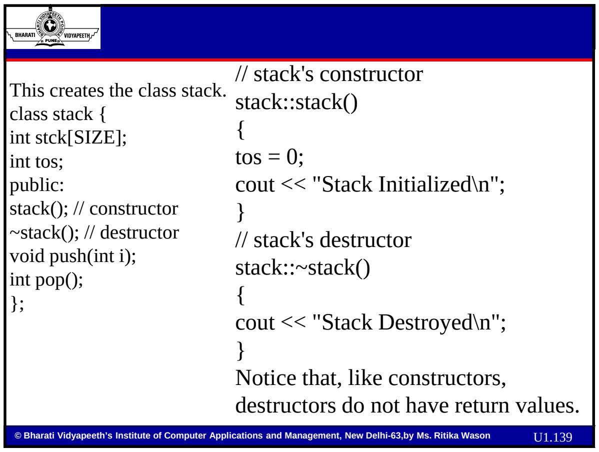
The complement of the constructor is the ***destructor.***

There are many reasons why a destructor may be needed. For example, an object may need to **deallocate memory that it had previously allocated or it may need to close a file that it had opened.**

In C++, it is the destructor that handles **deactivation events**.

The destructor has the same name as the constructor, but it is preceded by a ~

. For example, here is the stack class and its constructor and destructor.

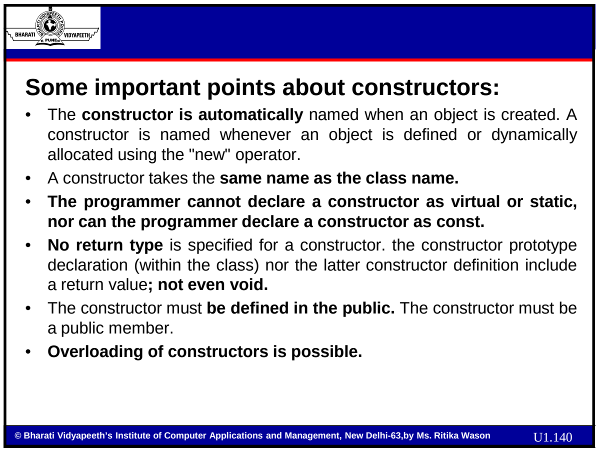


This creates the class stack.

```
class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
~stack(); // destructor
void push(int i);
int pop();
};
```

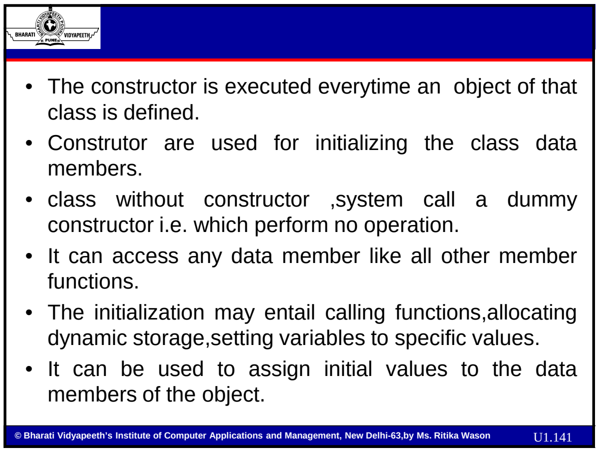
```
// stack's constructor
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
// stack's destructor
stack::~stack()
{
cout << "Stack Destroyed\n";
}
```

Notice that, like constructors, destructors do not have return values.



Some important points about constructors:

- The **constructor is automatically** named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.
- A constructor takes the **same name as the class name.**
- **The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const.**
- **No return type** is specified for a constructor. the constructor prototype declaration (within the class) nor the latter constructor definition include a return value**; not even void.**
- The constructor must **be defined in the public.** The constructor must be a public member.
- **Overloading of constructors is possible.**



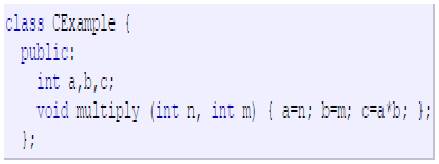
- The constructor is executed everytime an object of that class is defined.
- Construtor are used for initializing the class data members.
- class without constructor ,system call a dummy constructor i.e. which perform no operation.
- It can access any data member like all other member functions.
- The initialization may entail calling functions,allocating dynamic storage,setting variables to specific values.
- It can be used to assign initial values to the data members of the object.

- Each class can have only one default constructor.
- Default constructor is use to initialize the data which used by other member functions.

Constructor General Form:

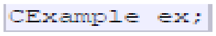
```
class class-name
{
   Access Specifier:
      Member-Variables
      Member-Functions
   public:
      class-name()
      {
         // Constructor code
      }
    .. other Variables & Functions
}
```

- **Default constructor**
- If you **do not declare any constructors** in a class definition, **the compiler assumes the class to have a default constructor with no arguments.** Therefore, after declaring a class like this one:



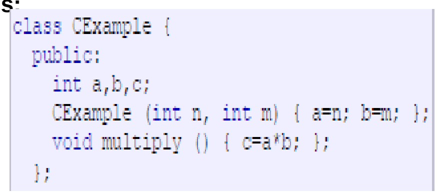
```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simplydeclaring them without any arguments:



```
CExample ex;
```

- But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. **So you have to declare all objects of that class according to the constructor prototypes you defined for the class:**



```
class CExample {
  public:
    int a,b,c;
    CExample (int n, int m) { a=n; b=m; };
    void multiply () { c=a*b; };
  };
```

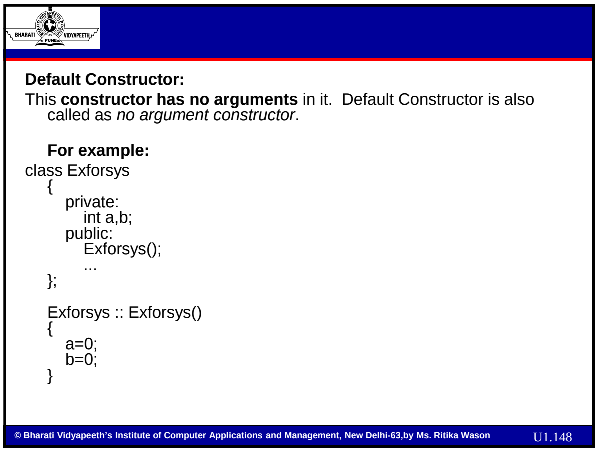
- Here we have declared a **constructor that takes two parameters of type int.** Therefore the following object declaration would be correct:

CExample ex (2,3);

But

CExample ex;

- **Would not be correct**, since we have declared the class to have an **explicit constructor**, **thus replacing the default constructor.**

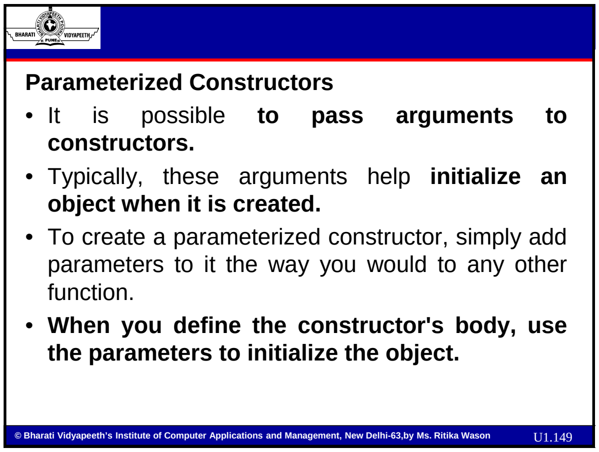


Default Constructor:

This **constructor has no arguments** in it. Default Constructor is also called as *no argument constructor*.

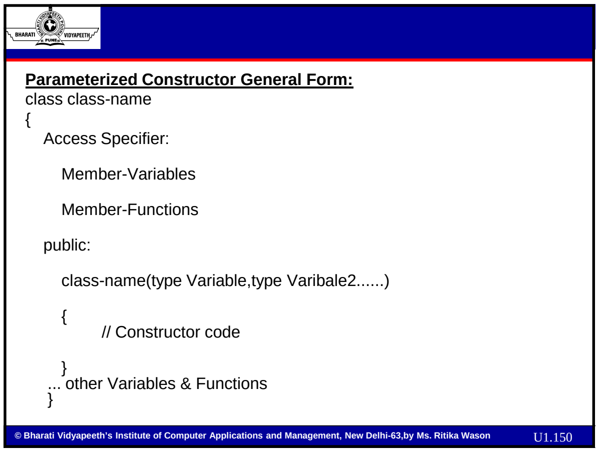
For example:

```
class Exforsys
   {
      private:
         int a,b;
      public:
         Exforsys();
         ...
   };

   Exforsys :: Exforsys()
   {
      a=0;
      b=0;
   }
```



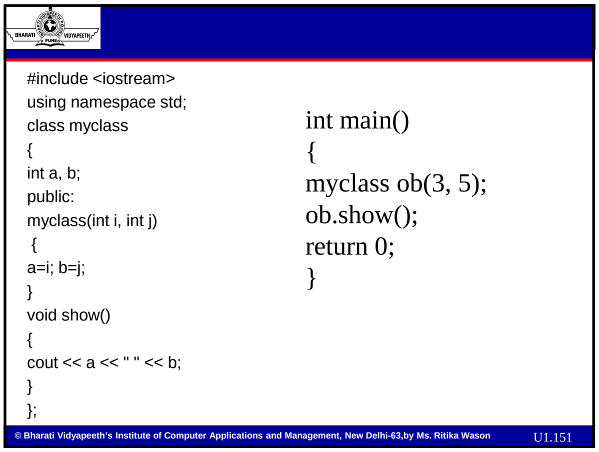
Parameterized Constructors

- It is possible **to pass arguments to constructors.**
- Typically, these arguments help **initialize an object when it is created.**
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- **When you define the constructor's body, use the parameters to initialize the object.**



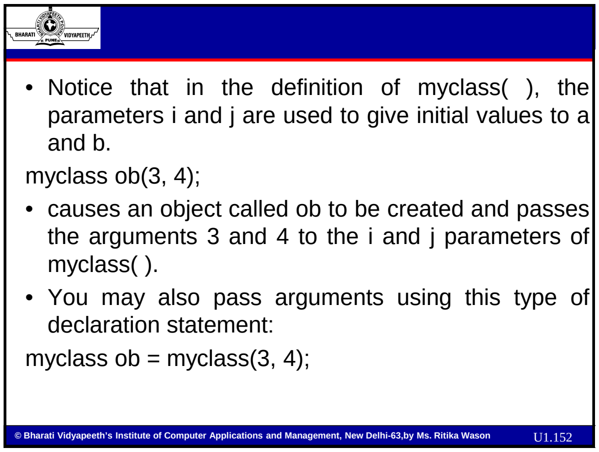
Parameterized Constructor General Form:

```
class class-name
{
   Access Specifier:

      Member-Variables

      Member-Functions

   public:

      class-name(type Variable,type Varibale2......)

      {
            // Constructor code

      }
   ... other Variables & Functions
   }
```



```
#include <iostream>
using namespace std;
class myclass
{
int a, b;
public:
myclass(int i, int j)
 {
a=i; b=j;
}
void show()
{
cout << a << " " << b;
}
};
```

```
int main()
{
myclass ob(3, 5);
ob.show();
return 0;
}
```

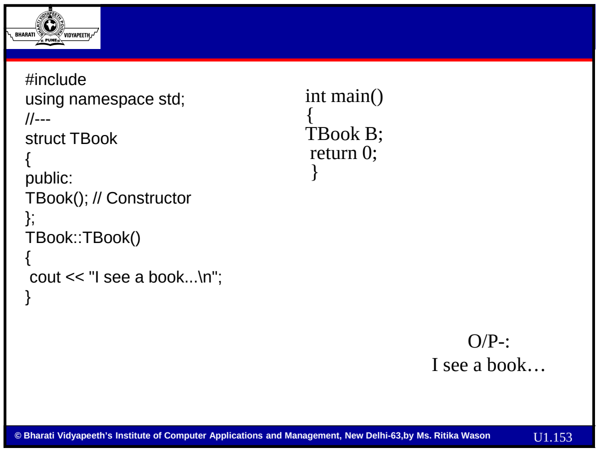


- Notice that in the definition of myclass( ), the parameters i and j are used to give initial values to a and b.

myclass ob(3, 4);

- causes an object called ob to be created and passes the arguments 3 and 4 to the i and j parameters of myclass( ).
- You may also pass arguments using this type of declaration statement:

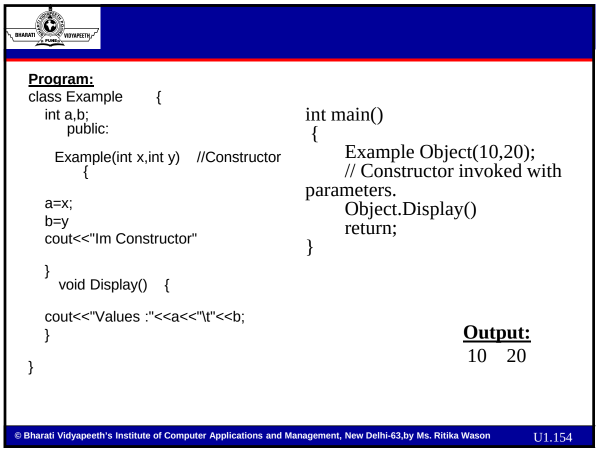
myclass ob = myclass(3, 4);



```
#include
using namespace std;
//---
struct TBook
{
public:
TBook(); // Constructor
};
TBook::TBook()
{
 cout << "I see a book...\n";
}
```

```
int main()
{
TBook B;
 return 0;
 }
```

O/P-:
I see a book…



Program:

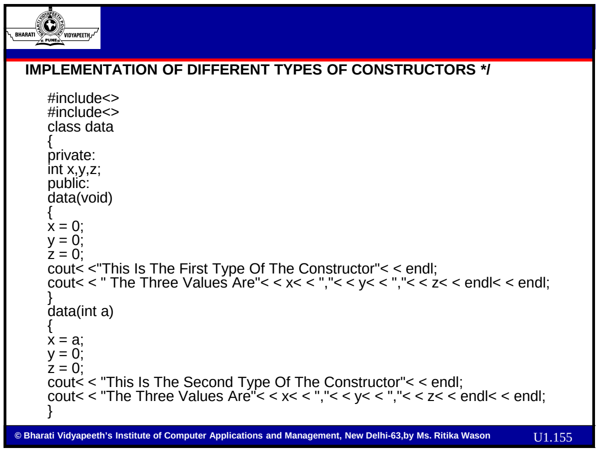
```
class Example       {
   int a,b;
       public:

     Example(int x,int y)    //Constructor
          {

   a=x;
   b=y
   cout<<"Im Constructor"

   }
      void Display()    {

   cout<<"Values :"<<a<<"\t"<<b;
   }

}
```

```
int main()
 {
        Example Object(10,20);
        // Constructor invoked with
parameters.
        Object.Display()
        return;
}
```

Output:

```
10    20
```

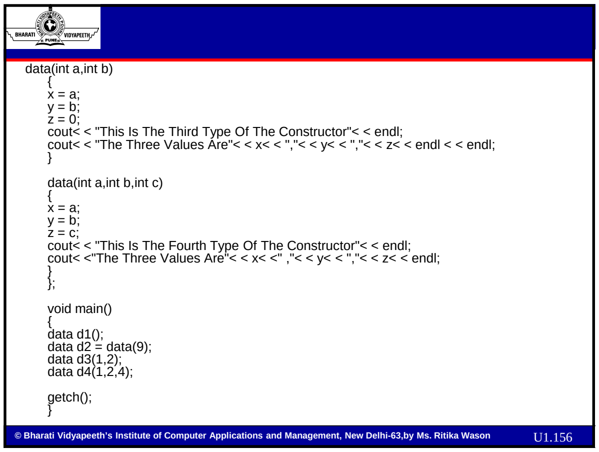
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.154



IMPLEMENTATION OF DIFFERENT TYPES OF CONSTRUCTORS */

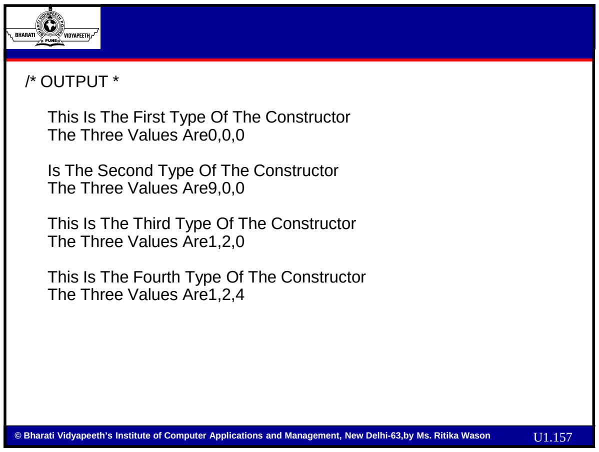
```
#include<>
#include<>
class data
{
private:
int x,y,z;
public:
data(void)
{
x = 0;
y = 0;
z = 0;
cout< <"This Is The First Type Of The Constructor"< < endl;
cout< < " The Three Values Are"< < x< < ","< < y< < ","< < z< < endl< < endl;
}
data(int a)
{
x = a;
y = 0;
z = 0;
cout< < "This Is The Second Type Of The Constructor"< < endl;
cout< < "The Three Values Are"< < x< < ","< < y< < ","< < z< < endl< < endl;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.155

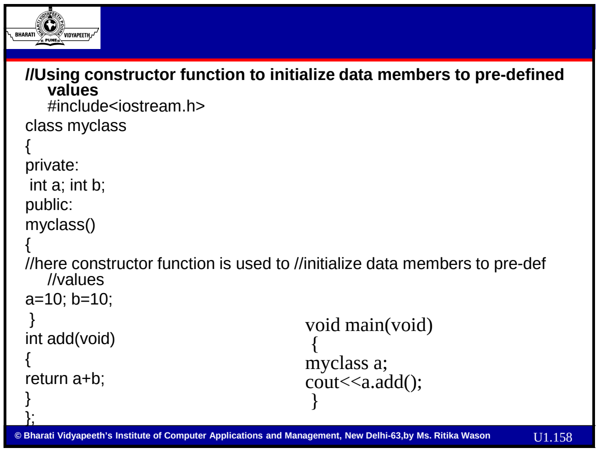


```
data(int a,int b)
    {
    x = a;
    y = b;
    z = 0;
    cout< < "This Is The Third Type Of The Constructor"< < endl;
    cout< < "The Three Values Are"< < x< < ","< < y< < ","< < z< < endl < < endl;
    }

    data(int a,int b,int c)
    {
    x = a;
    y = b;
    z = c;
    cout< < "This Is The Fourth Type Of The Constructor"< < endl;
    cout< <"The Three Values Are"< < x< <" ,"< < y< < ","< < z< < endl;
    }
    };

    void main()
    {
    data d1();
    data d2 = data(9);
    data d3(1,2);
    data d4(1,2,4);

    getch();
    }
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.156



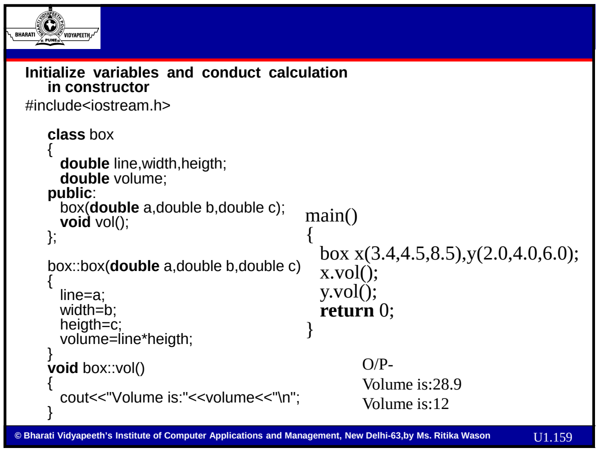
```
/* OUTPUT *

    This Is The First Type Of The Constructor
    The Three Values Are0,0,0

    Is The Second Type Of The Constructor
    The Three Values Are9,0,0

    This Is The Third Type Of The Constructor
    The Three Values Are1,2,0

    This Is The Fourth Type Of The Constructor
    The Three Values Are1,2,4
```



//Using constructor function to initialize data members to pre-defined values

```
    #include<iostream.h>
class myclass
{
private:
 int a; int b;
public:
myclass()
{
//here constructor function is used to //initialize data members to pre-def
    //values
a=10; b=10;
 }
int add(void)
{
return a+b;
}
};
```

```
void main(void)
 {
myclass a;
cout<<a.add();
 }
```

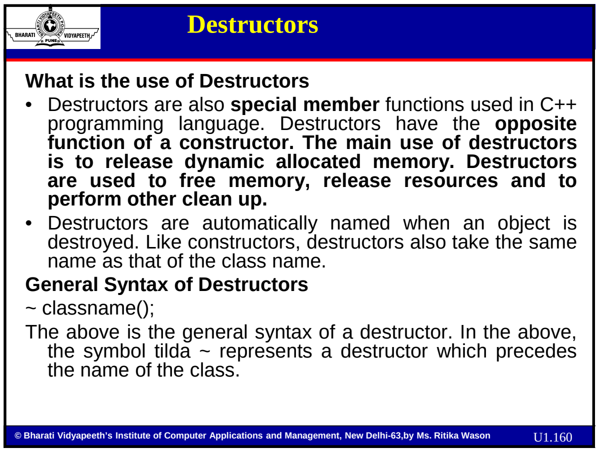


Initialize variables and conduct calculation in constructor

```
#include<iostream.h>

    class box
    {
      double line,width,heigth;
      double volume;
    public:
      box(double a,double b,double c);
      void vol();
    };

    box::box(double a,double b,double c)
    {
      line=a;
      width=b;
      heigth=c;
      volume=line*heigth;
    }
    void box::vol()
    {
      cout<<"Volume is:"<<volume<<"\n";
    }
```

```
main()
{
  box x(3.4,4.5,8.5),y(2.0,4.0,6.0);
  x.vol();
  y.vol();
  return 0;
}
```

```
O/P-
Volume is:28.9
Volume is:12
```



Destructors

What is the use of Destructors

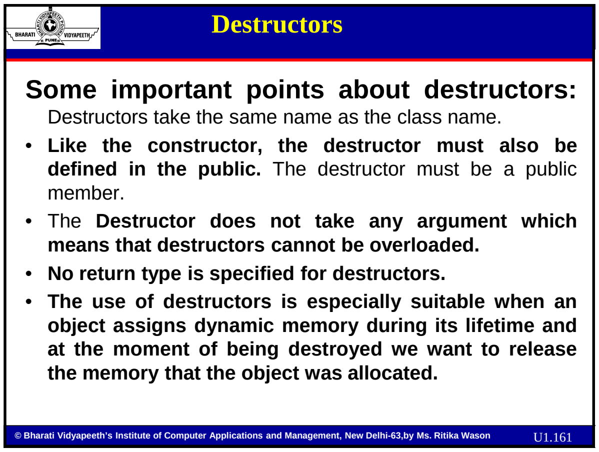
- Destructors are also **special member** functions used in C++ programming language. Destructors have the **opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up.**
- Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.160



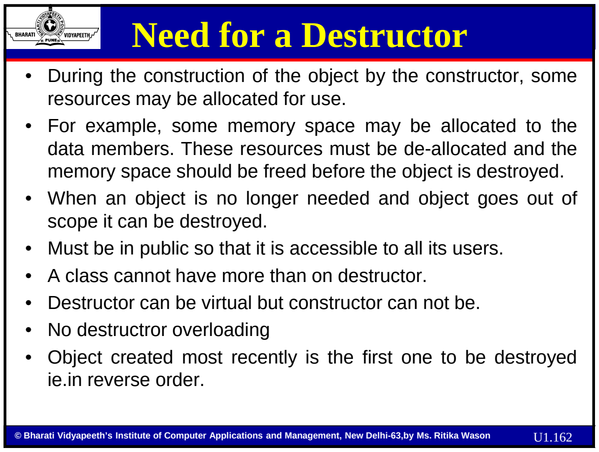
Destructors

Some important points about destructors:

Destructors take the same name as the class name.

- **Like the constructor, the destructor must also be defined in the public.** The destructor must be a public member.
- The **Destructor does not take any argument which means that destructors cannot be overloaded.**
- **No return type is specified for destructors.**
- **The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.161



Need for a Destructor

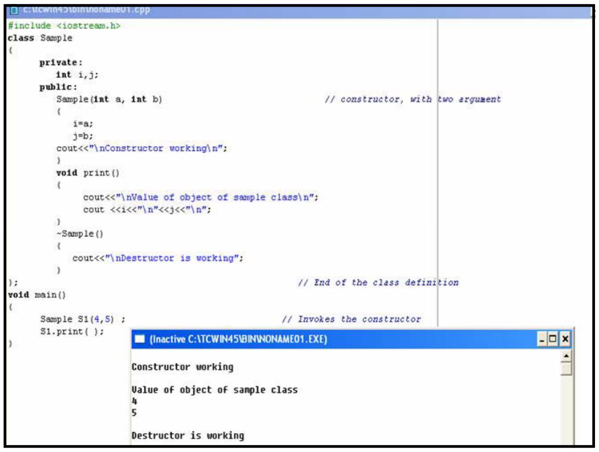
- During the construction of the object by the constructor, some resources may be allocated for use.
- For example, some memory space may be allocated to the data members. These resources must be de-allocated and the memory space should be freed before the object is destroyed.
- When an object is no longer needed and object goes out of scope it can be destroyed.
- Must be in public so that it is accessible to all its users.
- A class cannot have more than on destructor.
- Destructor can be virtual but constructor can not be.
- No destructror overloading
- Object created most recently is the first one to be destroyed ie.in reverse order.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.162

```
class Exforsys
   {
      private:
         ……………
      public:
         Exforsys()
         { }
         ~ Exforsys()
         { }
   }
```

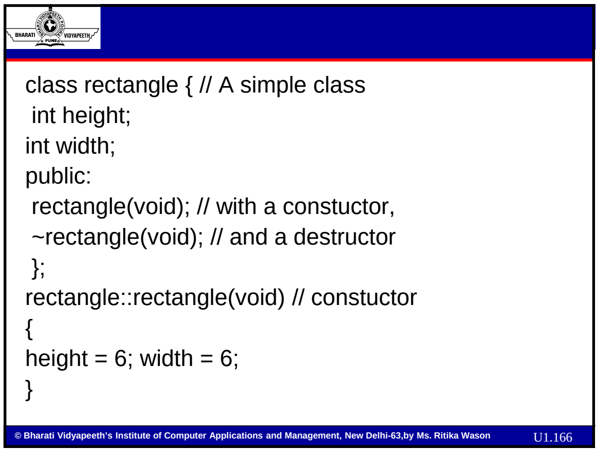
Destructors

- Whenever an object is created within a program, it also needs to be destroyed.
- **If a class has constructor to initialize members, it should also have a destructor to free up the used memory.**
- **A destructor, as the name suggest, destroys the values of the object created by the constructor when the object goes out of scope.**
- A destructor is also a member function whose name is the same name as that of a class, but is preceded by tilde (‘**~**`).
- For example, the destructor of class **salesperson** will be **~salesperson()**.
- **A destructor does not take any arguments nor does it return any value. The compiler automatically calls them when the objects are destroyed.**

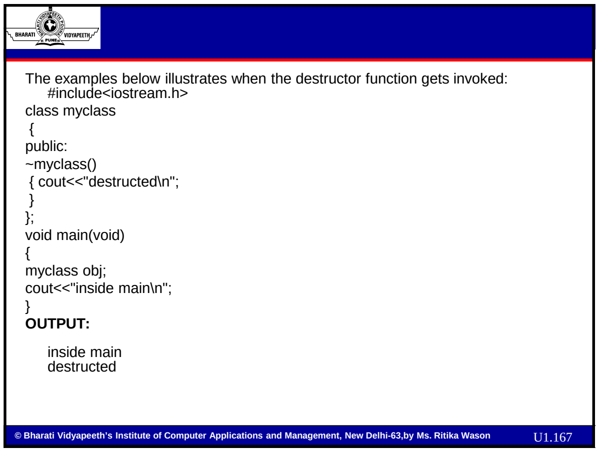

C:\tcwin45\bin\noname01.cpp
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
#include <iostream.h>
class Sample
{
      private:
         int i,j;
      public:
         Sample(int a, int b)                           // constructor, with two argument
         {
            i=a;
            j=b;
         cout<<"\nConstructor working\n";
         }
         void print()
         {
              cout<<"\nValue of object of sample class\n";
              cout <<i<<"\n"<<j<<"\n";
         }
         ~Sample()
         {
            cout<<"\nDestructor is working";
         }
};                                                     // End of the class definition
void main()
{
      Sample S1(4,5) ;                                // Invokes the constructor
      S1.print( );
}
```

```
Constructor working

Value of object of sample class
4
5

Destructor is working
```



```
class rectangle { // A simple class
 int height;
int width;
public:
 rectangle(void); // with a constuctor,
 ~rectangle(void); // and a destructor
 };
rectangle::rectangle(void) // constuctor
{
height = 6; width = 6;
}
```

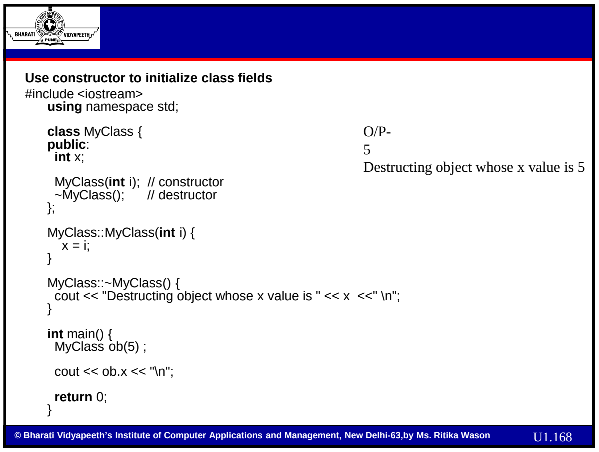


The examples below illustrates when the destructor function gets invoked:

```
    #include<iostream.h>
class myclass
 {
public:
~myclass()
 { cout<<"destructed\n";
 }
};
void main(void)
{
myclass obj;
cout<<"inside main\n";
}
```

OUTPUT:

```
    inside main
    destructed
```

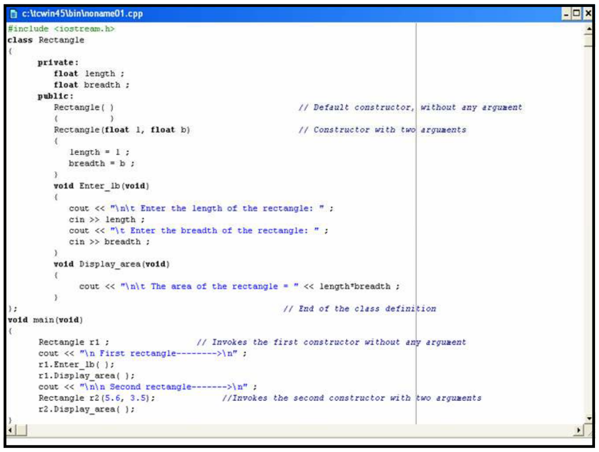


Use constructor to initialize class fields

```
#include <iostream>
    using namespace std;

    class MyClass {
    public:
      int x;

      MyClass(int i);  // constructor
      ~MyClass();      // destructor
    };

    MyClass::MyClass(int i) {
        x = i;
    }

    MyClass::~MyClass() {
      cout << "Destructing object whose x value is " << x  <<" \n";
    }

    int main() {
      MyClass ob(5) ;

      cout << ob.x << "\n";

      return 0;
    }
```

O/P-

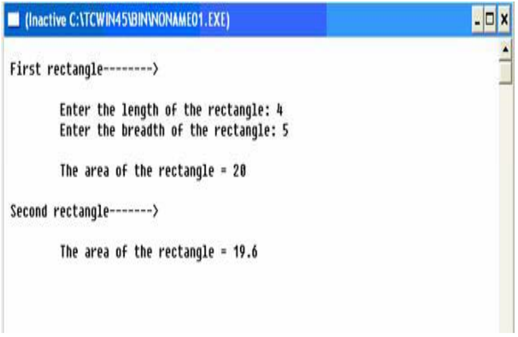
```
5
Destructing object whose x value is 5
```



c:\tcwin45\bin\noname01.cpp

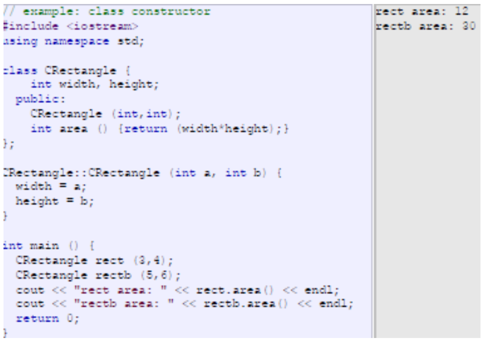
```
#include <iostream.h>
class Rectangle
{
      private:
         float length ;
         float breadth ;
      public:
         Rectangle( )                                // Default constructor, without any argument
         {          }
         Rectangle(float l, float b)                 // Constructor with two arguments
         {
            length = l ;
            breadth = b ;
         }
         void Enter_lb(void)
         {
            cout << "\n\t Enter the length of the rectangle: " ;
            cin >> length ;
            cout << "\t Enter the breadth of the rectangle: " ;
            cin >> breadth ;
         }
         void Display_area(void)
         {
              cout << "\n\t The area of the rectangle = " << length*breadth ;
         }
};                                                 // End of the class definition
void main(void)
{
      Rectangle r1 ;                 // Invokes the first constructor without any argument
      cout << "\n First rectangle-------->\n" ;
      r1.Enter_lb( );
      r1.Display_area( );
      cout << "\n\n Second rectangle------->\n" ;
      Rectangle r2(5.6, 3.5);             //Invokes the second constructor with two arguments
      r2.Display_area( );
}
```

Output



(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
First rectangle-------->

        Enter the length of the rectangle: 4
        Enter the breadth of the rectangle: 5

        The area of the rectangle = 20

Second rectangle------->

        The area of the rectangle = 19.6
```



```
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

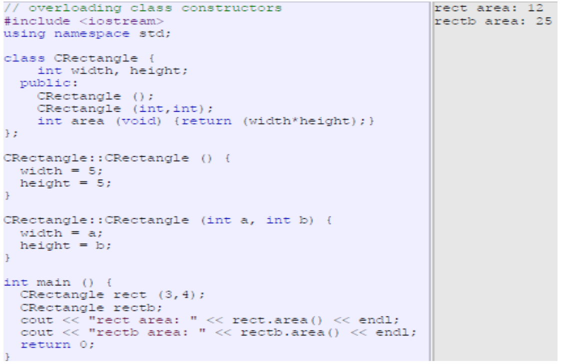
```
rect area: 12
rectb area: 30
```

Overloading Constructors

Overloading Constructors

•Like any other function, a **constructor can also be overloaded** with more than one function **that have the same name but different types or number of parameters.**

•Remember that for overloaded functions the **compiler will call the one whose parameters match the arguments used in the function call.**

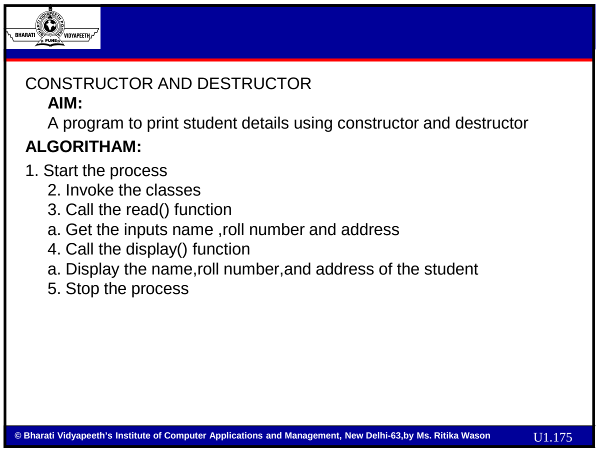


```
// overloading class constructors
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

Important: Notice how if we declare a new object and **we want to use its default constructor** (the one without parameters), **we do not include parentheses ():**

CRectangle rectb; // right
CRectangle rectb(); // wrong!



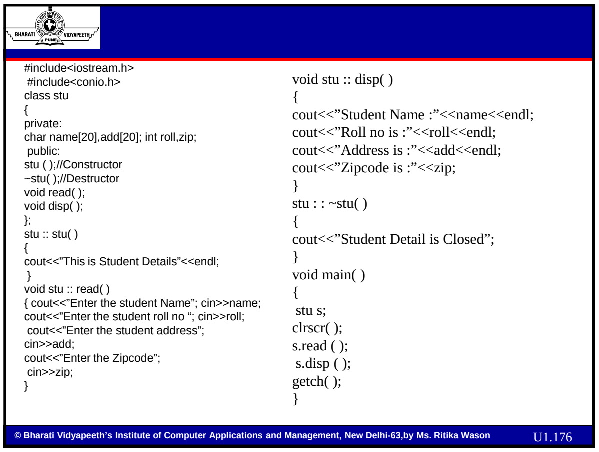
CONSTRUCTOR AND DESTRUCTOR

AIM:

A program to print student details using constructor and destructor

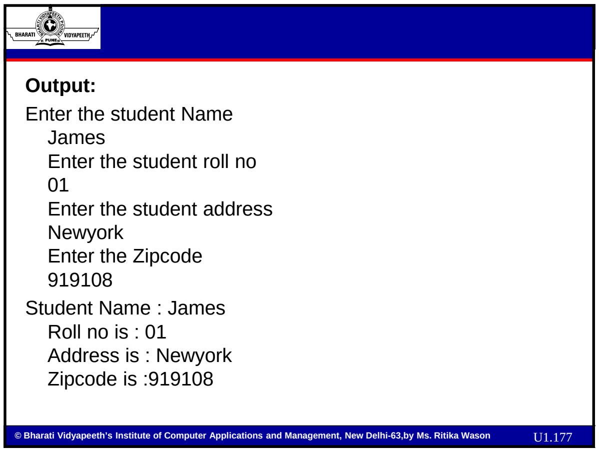
ALGORITHAM:

```
1. Start the process
    2. Invoke the classes
    3. Call the read() function
    a. Get the inputs name ,roll number and address
    4. Call the display() function
    a. Display the name,roll number,and address of the student
    5. Stop the process
```



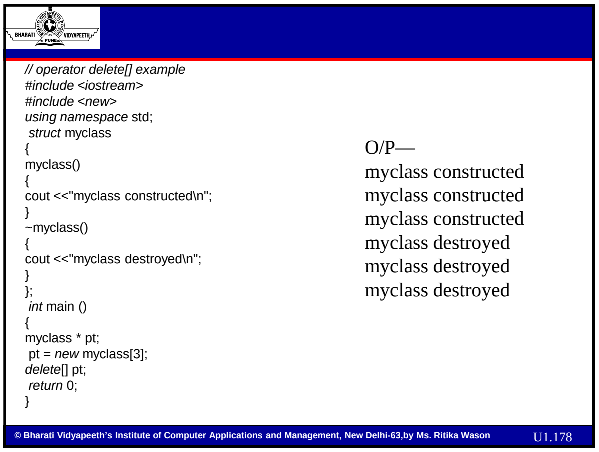
```
#include<iostream.h>
 #include<conio.h>
class stu
{
private:
char name[20],add[20]; int roll,zip;
 public:
stu ( );//Constructor
~stu( );//Destructor
void read( );
void disp( );
};
stu :: stu( )
{
cout<<"This is Student Details"<<endl;
 }
void stu :: read( )
{ cout<<"Enter the student Name"; cin>>name;
cout<<"Enter the student roll no "; cin>>roll;
 cout<<"Enter the student address";
cin>>add;
cout<<"Enter the Zipcode";
 cin>>zip;
}
```

```
void stu :: disp( )
{
cout<<"Student Name :"<<name<<endl;
cout<<"Roll no is :"<<roll<<endl;
cout<<"Address is :"<<add<<endl;
cout<<"Zipcode is :"<<zip;
}
stu : : ~stu( )
{
cout<<"Student Detail is Closed";
}
void main( )
{
 stu s;
clrscr( );
s.read ( );
 s.disp ( );
getch( );
}
```



Output:

```
Enter the student Name
    James
    Enter the student roll no
    01
    Enter the student address
    Newyork
    Enter the Zipcode
    919108
Student Name : James
    Roll no is : 01
    Address is : Newyork
    Zipcode is :919108
```

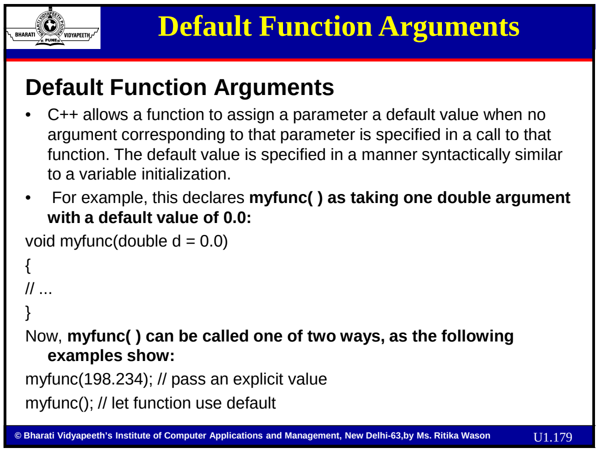


```
// operator delete[] example
#include <iostream>
#include <new>
using namespace std;
 struct myclass
{
myclass()
{
cout <<"myclass constructed\n";
}
~myclass()
{
cout <<"myclass destroyed\n";
}
};
 int main ()
{
myclass * pt;
 pt = new myclass[3];
delete[] pt;
 return 0;
}
```

O/P—

```
myclass constructed
myclass constructed
myclass constructed
myclass destroyed
myclass destroyed
myclass destroyed
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.178



Default Function Arguments

Default Function Arguments

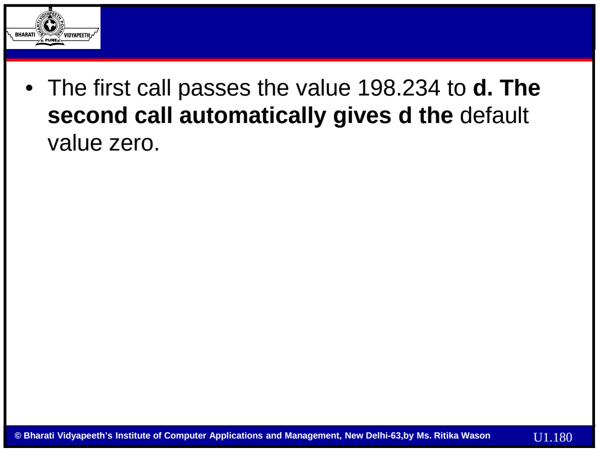
- C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization.
- For example, this declares **myfunc( ) as taking one double argument with a default value of 0.0:**

```
void myfunc(double d = 0.0)
{
// ...
}
```

Now, **myfunc( ) can be called one of two ways, as the following examples show:**

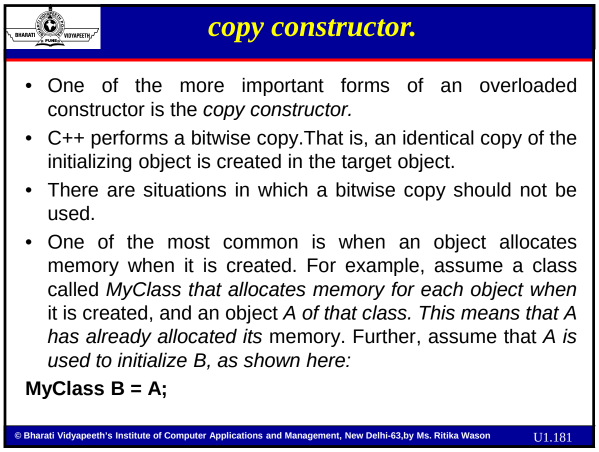
```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.179



- The first call passes the value 198.234 to **d. The second call automatically gives d the** default value zero.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.180

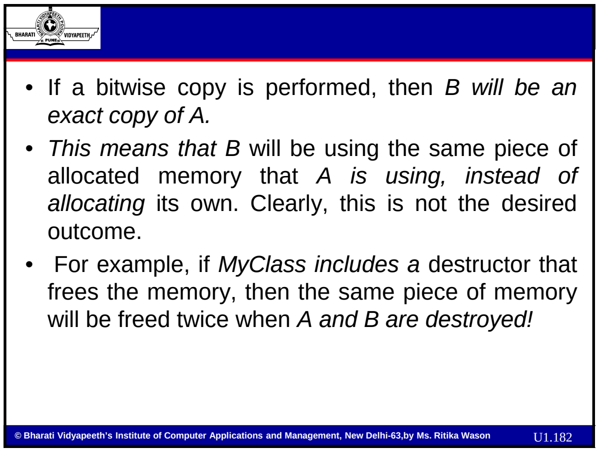


copy constructor.

- One of the more important forms of an overloaded constructor is the *copy constructor.*
- C++ performs a bitwise copy.That is, an identical copy of the initializing object is created in the target object.
- There are situations in which a bitwise copy should not be used.
- One of the most common is when an object allocates memory when it is created. For example, assume a class called *MyClass that allocates memory for each object when* it is created, and an object *A of that class. This means that A has already allocated its* memory. Further, assume that *A is used to initialize B, as shown here:*

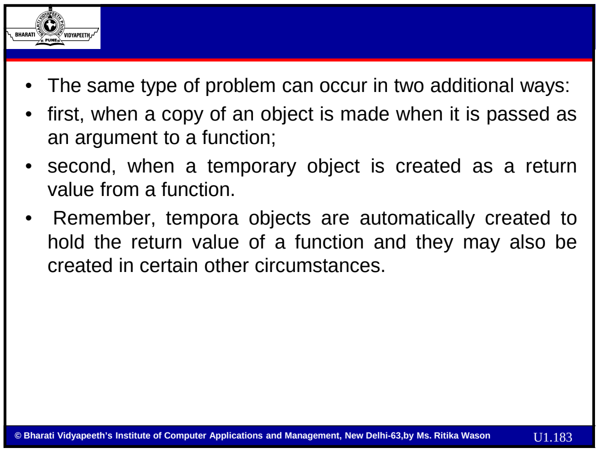
MyClass B = A;

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.181



- If a bitwise copy is performed, then *B will be an exact copy of A.*
- *This means that B* will be using the same piece of allocated memory that *A is using, instead of allocating* its own. Clearly, this is not the desired outcome.
- For example, if *MyClass includes a* destructor that frees the memory, then the same piece of memory will be freed twice when *A and B are destroyed!*

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.182



- The same type of problem can occur in two additional ways:
- first, when a copy of an object is made when it is passed as an argument to a function;
- second, when a temporary object is created as a return value from a function.
- Remember, tempora objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.183

C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy. The most common general form of a copy constructor is

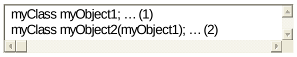
classname (*classname &o) {*

// body of constructor

}

Copy Constructor

- A copy constructor is a class constructor that can be used to **initialize one object with the values of another object** of the same class during the declaration statement.
- **In simple words, we can say that, if we have an object called myObject1 and we want to create a new object called myObject2, initialized with the contents of myObject1 then the copy constructor should be used.**
- Consider the declaration given below:



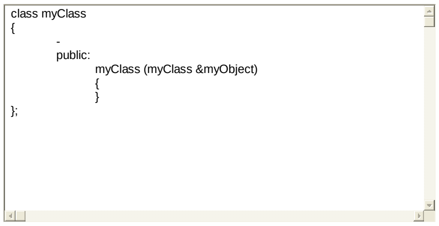
```
myClass myObject1; ... (1)
myClass myObject2(myObject1); ... (2)
```

•The declaration (1) declares an object **myObject1 of class myClass.**

•The declaration (2) declares another object **myObject2** of class myClass as well as initializes it with the contents of existing object myObject1.

Copy Constructor

- The copy constructor is, however, **defined in the class as a parameterized constructor receiving an object as argument passed by reference.**



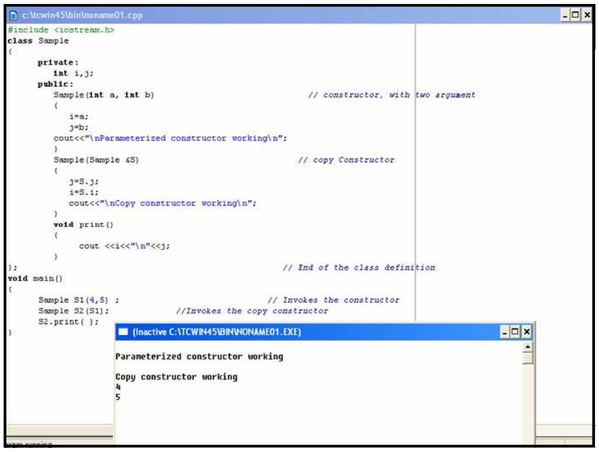
```
class myClass
{
            -
            public:
                        myClass (myClass &myObject)
                        {
                        }
};
```

Copy Constructor

- A copy constructor is a constructor of the form **classname(classname &)**.
- The compiler will use the copy constructor whenever you initialize an instance, using values of another instance of the same type.
- Example:

```
salesperson S1 ; // default constructor used
```

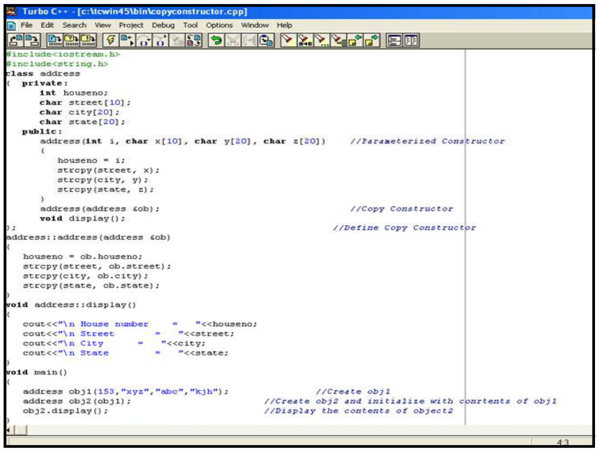
- The second statement will copy the instance of S1 to S2.


c:\tcwin45\bin\noname01.cpp

```
#include <iostream.h>
class Sample
{
      private:
         int i,j;
      public:
         Sample(int a, int b)                              // constructor, with two argument
         {
            i=a;
            j=b;
         cout<<"\nParameterized constructor working\n";
         }
         Sample(Sample &S)                                // copy Constructor
         {
            j=S.j;
            i=S.i;
            cout<<"\nCopy constructor working\n";
         }
         void print()
         {
              cout <<i<<"\n"<<j;
         }
};                                                       // End of the class definition
void main()
{
      Sample S1(4,5) ;                                 // Invokes the constructor
      Sample S2(S1);                 //Invokes the copy constructor
      S2.print( );
}
```

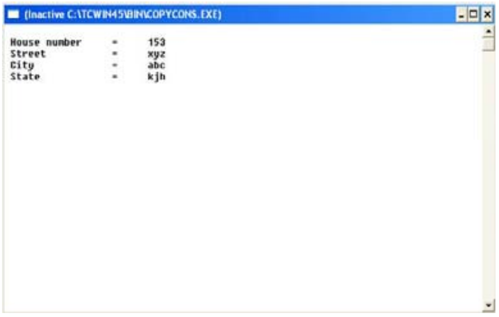
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
Parameterized constructor working

Copy constructor working
4
5
```



```
#include<iostream.h>
#include<string.h>
class address
{  private:
      int houseno;
      char street[10];
      char city[20];
      char state[20];
   public:
      address(int i, char x[10], char y[20], char z[20])    //Parameterized Constructor
      {
         houseno = i;
         strcpy(street, x);
         strcpy(city, y);
         strcpy(state, z);
      }
      address(address &ob);                                 //Copy Constructor
      void display();
};                                                        //Define Copy Constructor
address::address(address &ob)
{
   houseno = ob.houseno;
   strcpy(street, ob.street);
   strcpy(city, ob.city);
   strcpy(state, ob.state);
}
void address::display()
{
   cout<<"\n House number    =   "<<houseno;
   cout<<"\n Street        =   "<<street;
   cout<<"\n City       =   "<<city;
   cout<<"\n State         =   "<<state;
}
void main()
{
   address obj1(153,"xyz","abc","kjh");                //Create obj1
   address obj2(obj1);                       //Create obj2 and initialize with conrtents of obj1
   obj2.display();                           //Display the contents of object2
}
```

Output



```
House number     =      153
Street           =      xyz
City             =      abc
State            =      kjh
```

Explanation

•An object called **obj1** of class **address** is created with some initial values. Another object **obj2**, a copy of obj1, is created with the help of the copy constructor.

•At this stage, a question which arises is that, this job could have been done by simple assignment of objects i.e. **obj1 = obj2,** then why to use a copy constructor?

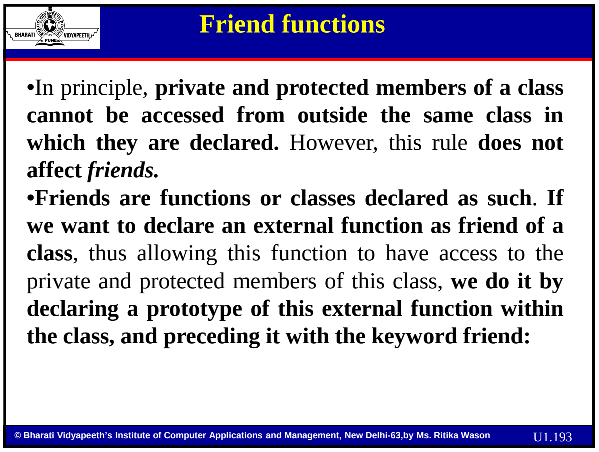
•The need of a copy constructor is felt when the class includes pointers which need to be properly initialized. A simple assignment will fail to do this, because both the copies will hold pointers to same memory location.

- But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own.
- These are the *copy constructor, the copy assignment operator, and the default destructor.*
- The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object.
- For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (CExample& rv)
{
a=rv.a; b=rv.b; c=rv.c;
}
```

Therefore, the two following object declarations would be correct:

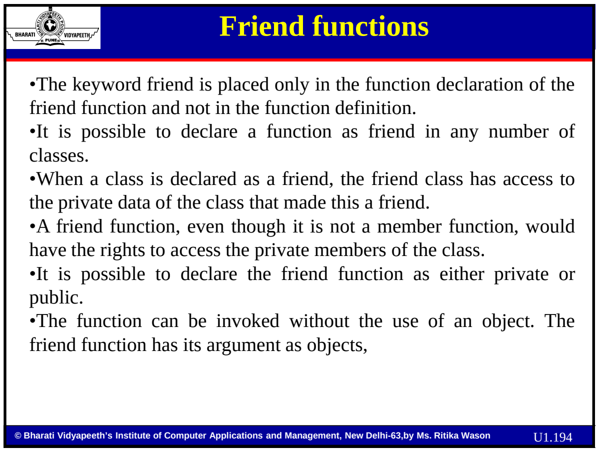
```
CExample ex (2,3);
CExample ex2 (ex); // copy constructor (data copied from ex)
```



Friend functions

•In principle, **private and protected members of a class cannot be accessed from outside the same class in which they are declared.** However, this rule **does not affect *friends.***

•**Friends are functions or classes declared as such. If we want to declare an external function as friend of a class**, thus allowing this function to have access to the private and protected members of this class, **we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:**



Friend functions

•The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

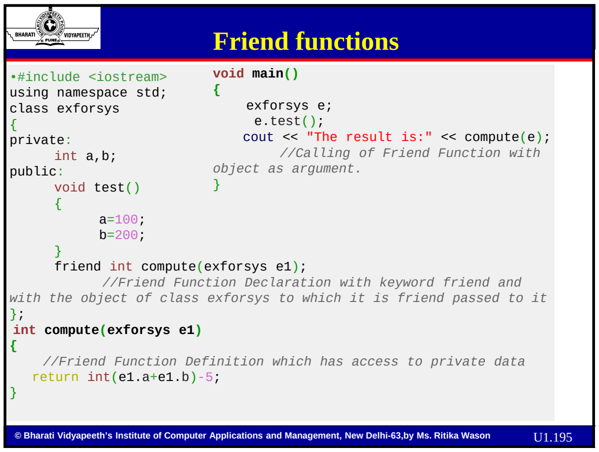
•It is possible to declare a function as friend in any number of classes.

•When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.

•A friend function, even though it is not a member function, would have the rights to access the private members of the class.

•It is possible to declare the friend function as either private or public.

•The function can be invoked without the use of an object. The friend function has its argument as objects,



Friend functions

```
•#include <iostream>
using namespace std;
class exforsys
{
private:
      int a,b;
public:
      void test()
      {
            a=100;
            b=200;
      }
      friend int compute(exforsys e1);
            //Friend Function Declaration with keyword friend and
with the object of class exforsys to which it is friend passed to it
};
 int compute(exforsys e1)
{
    //Friend Function Definition which has access to private data
   return int(e1.a+e1.b)-5;
}
```

```
void main()
{
    exforsys e;
     e.test();
    cout << "The result is:" << compute(e);
         //Calling of Friend Function with
object as argument.
}
```

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (**.** and –>) unless they are members of another class.

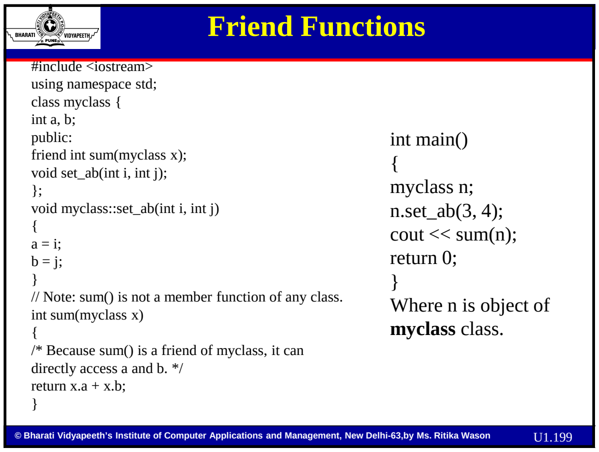
Friend Functions

- It is possible to grant a nonmember function access to the private members of a class by using a **friend.**
- **A friend function has access to all private and protected members** of the class for which it is a **friend.**
- **To declare a friend function, include its prototype** within the class, preceding it with the keyword **friend.**
- Friend function **Can be declared in the private part or the public part.**
- Can be used to **bridge the two classes**.
- If a program uses many friend functions, it can easily be concluded that there is a basic flaw in the design of the program and it would be better to redesign such programs.

```
class  base
{
   int val1,val2;
  public:
   void get()
   {
      cout<<"Enter two values:";
      cin>>val1>>val2;
   }
   friend float mean(base ob);
};
float mean(base ob)
{
  return float(ob.val1+ob.val2)/2;
}
void main()
{
   clrscr();
   base obj;
   obj.get();
   cout<<"\n Mean value is : "<<mean(obj);
   getch();
}
```

Output:

```
Enter two values: 10, 20
Mean Value is: 15
```

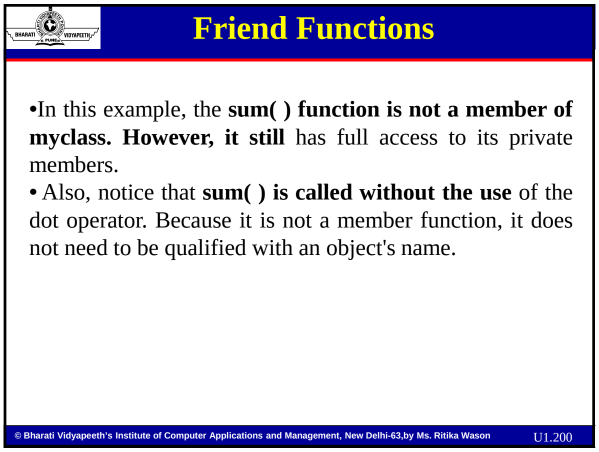


Friend Functions

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
// Note: sum() is not a member function of any class.
int sum(myclass x)
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}
```

```
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

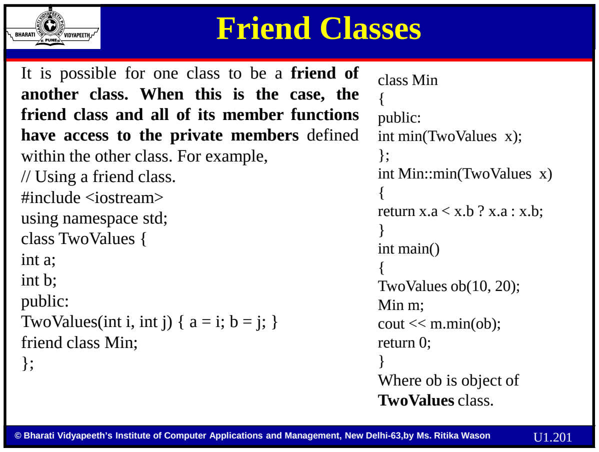
Where n is object of **myclass** class.



Friend Functions

•In this example, the **sum( ) function is not a member of myclass. However, it still** has full access to its private members.

• Also, notice that **sum( ) is called without the use** of the dot operator. Because it is not a member function, it does not need to be qualified with an object's name.



Friend Classes

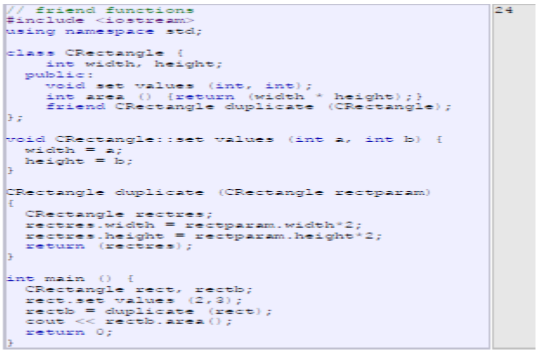
It is possible for one class to be a **friend of another class. When this is the case, the friend class and all of its member functions have access to the private members** defined within the other class. For example,

```
// Using a friend class.
#include <iostream>
using namespace std;
class TwoValues {
int a;
int b;
public:
TwoValues(int i, int j) { a = i; b = j; }
friend class Min;
};
```

```
class Min
{
public:
int min(TwoValues  x);
};
int Min::min(TwoValues  x)
{
return x.a < x.b ? x.a : x.b;
}
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
return 0;
}
```

Where ob is object of **TwoValues** class.

- In this example, class **Min has access to the private variables a and b declared within** the **TwoValues class.**
- It is critical to understand that when one class is a **friend of another, it only has** access to names defined within the other class. It does not inherit the other class.Specifically, the members of the first class do not become members of the **friend class.**



```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area () {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

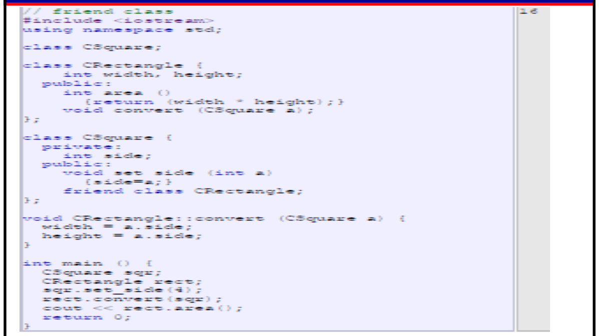
```
24
```

- The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members.
- Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

Friend classes

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.



```
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

16

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of Rectangle.

Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so Crectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have

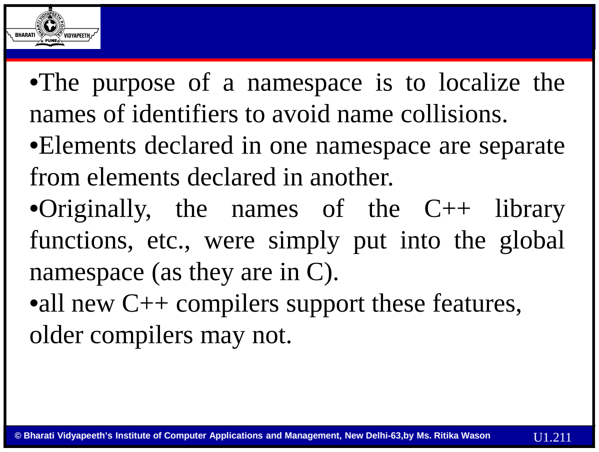
declared also CSquare as friend of CRectangle if we wanted to. Another property of friendships is that they are *not transitive: The friend of a friend is not considered to be a friend* unless explicitly specified

Example To Develop An Employee Class

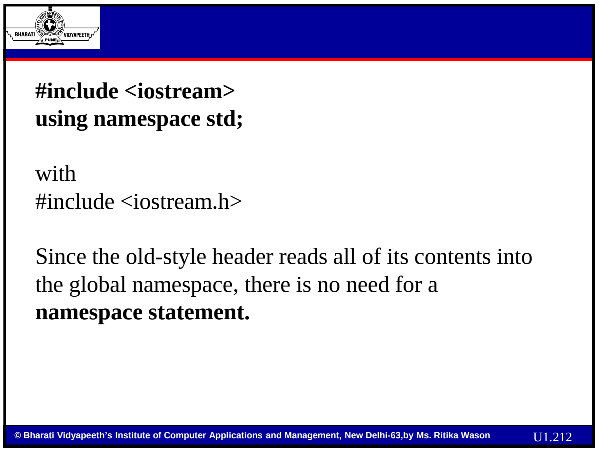
```
void main()
{
employee a1, b1;
a1.getdata(); b1.getdata();
float average = a1.calc(b1);
/* Object a1 invokes function calc(), b1 is passed as the parameter*/
cout<<endl<<“Average Salary:” <<average;
} // end of program.
```

using namespace std;

This tells the compiler to use the **std namespace. Namespaces are a recent addition** to C++. **A namespace creates a declarative region in which various program elements can be placed.** Namespaces help in the organization of large programs. The **using statement** informs the compiler that you want to use the **std namespace. This is the namespace in which the entire Standard C++ library is declared.** By using the std namespace you simplify access to the standard library. The programs in Part One, which use only the C subset, don't need a namespace statement because the C library functions are also available in the default, global namespace.



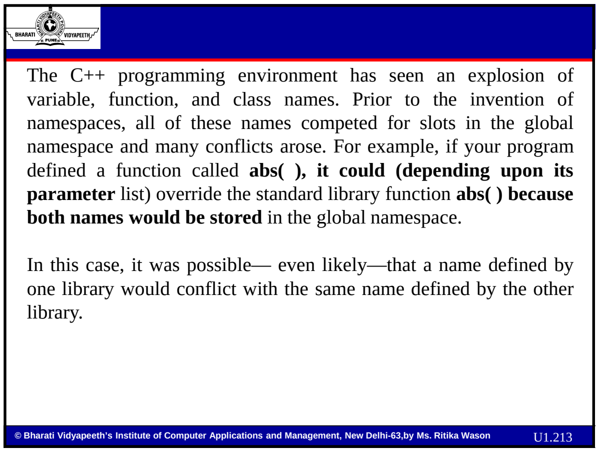
- The purpose of a namespace is to localize the names of identifiers to avoid name collisions.
- Elements declared in one namespace are separate from elements declared in another.
- Originally, the names of the C++ library functions, etc., were simply put into the global namespace (as they are in C).
- all new C++ compilers support these features, older compilers may not.



#include <iostream>
using namespace std;

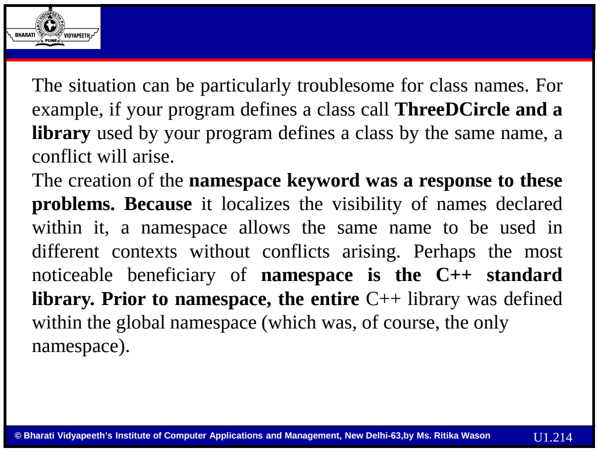
with
#include <iostream.h>

Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace statement.**



The C++ programming environment has seen an explosion of variable, function, and class names. Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose. For example, if your program defined a function called **abs( ), it could (depending upon its parameter** list) override the standard library function **abs( ) because both names would be stored** in the global namespace.

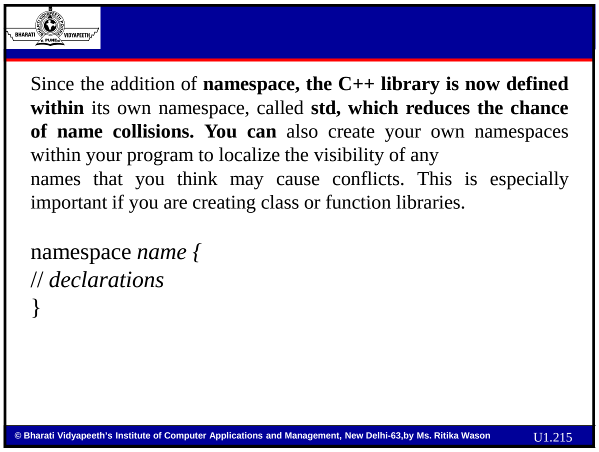
In this case, it was possible— even likely—that a name defined by one library would conflict with the same name defined by the other library.



The situation can be particularly troublesome for class names. For example, if your program defines a class call **ThreeDCircle and a library** used by your program defines a class by the same name, a conflict will arise.

The creation of the **namespace keyword was a response to these problems. Because** it localizes the visibility of names declared within it, a namespace allows the same name to be used in different contexts without conflicts arising. Perhaps the most noticeable beneficiary of **namespace is the C++ standard library. Prior to namespace, the entire** C++ library was defined within the global namespace (which was, of course, the only namespace).

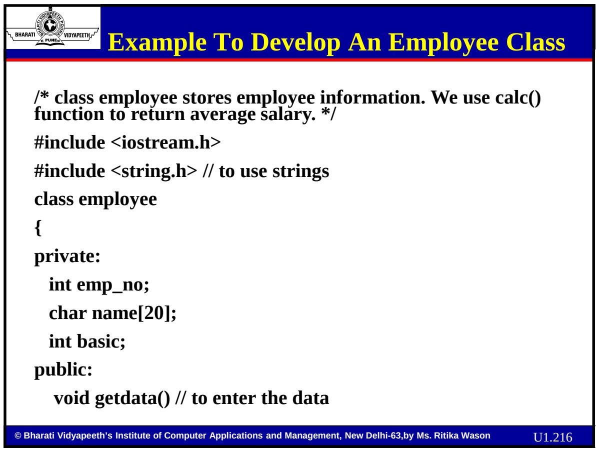
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.214



Since the addition of **namespace, the C++ library is now defined within** its own namespace, called **std, which reduces the chance of name collisions. You can** also create your own namespaces within your program to localize the visibility of any names that you think may cause conflicts. This is especially important if you are creating class or function libraries.

namespace *name {*
// *declarations*
}

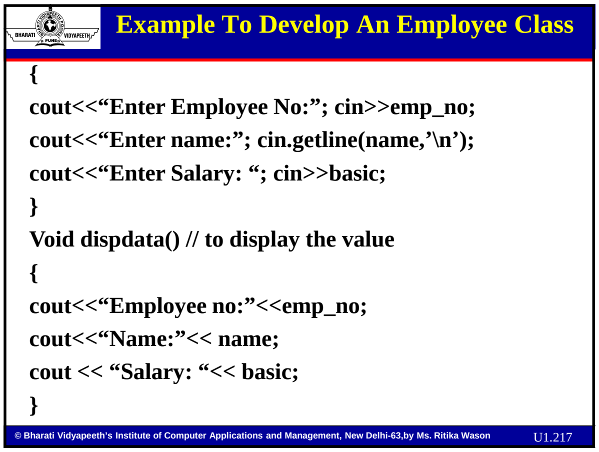
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.215



Example To Develop An Employee Class

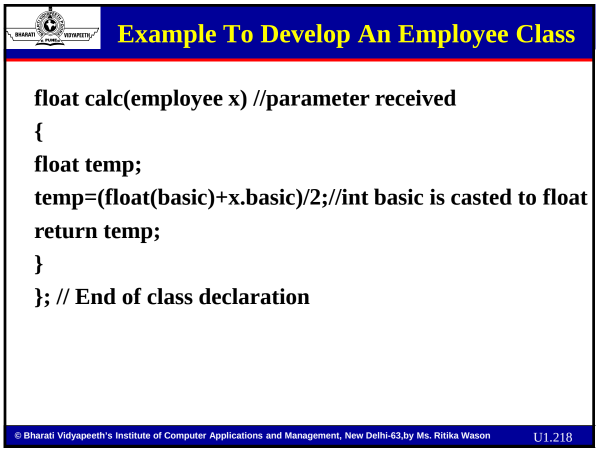
```
/* class employee stores employee information. We use calc()
function to return average salary. */
#include <iostream.h>
#include <string.h> // to use strings
class employee
{
private:
  int emp_no;
  char name[20];
  int basic;
public:
   void getdata() // to enter the data
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.216



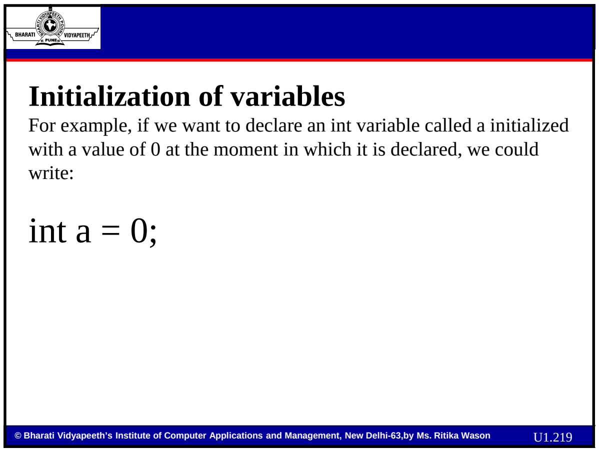
Example To Develop An Employee Class

```
{
cout<<“Enter Employee No:”; cin>>emp_no;
cout<<“Enter name:”; cin.getline(name,’\n’);
cout<<“Enter Salary: “; cin>>basic;
}
Void dispdata() // to display the value
{
cout<<“Employee no:”<<emp_no;
cout<<“Name:”<< name;
cout << “Salary: “<< basic;
}
```



Example To Develop An Employee Class

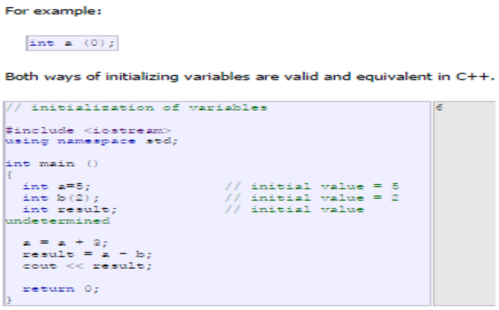
```
float calc(employee x) //parameter received
{
float temp;
temp=(float(basic)+x.basic)/2;//int basic is casted to float
return temp;
}
}; // End of class declaration
```



Initialization of variables

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```



For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;                     // initial value = 5
  int b(2);                    // initial value = 2
  int result;                  // initial value
undetermined

  a = a + 3;
  result = a - b;
  cout << result;

  return 0;
}
```

```
6
```

Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memoryconsuming variables, simply by using the #define preprocessor directive. Its format is:

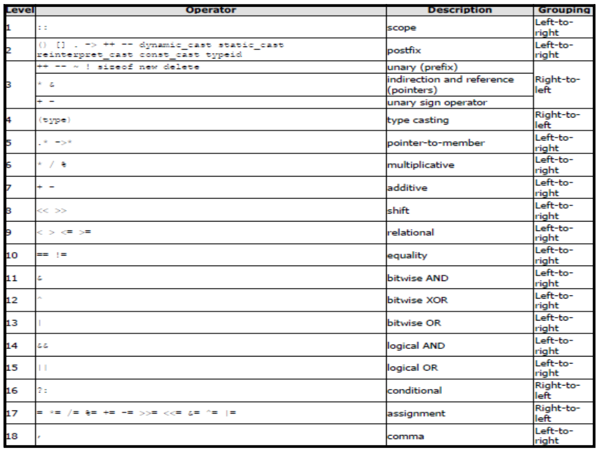
#define identifier value

For example:

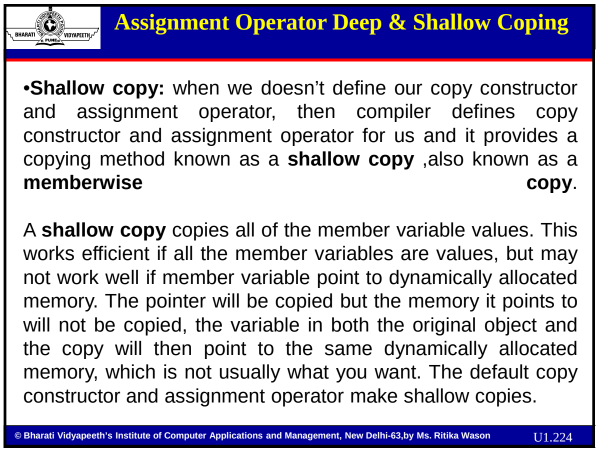
#define PI 3.14159

#define NEWLINE '\n'

```
// defined constants: calculate circumference
#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << circle;
cout << NEWLINE;
return 0;
}
```



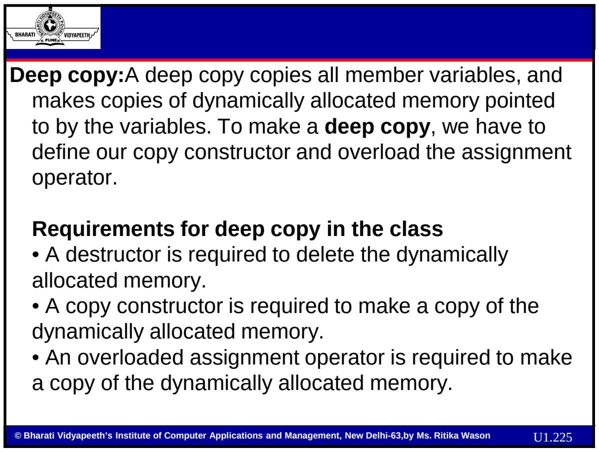
Level	Operator	Description	Grouping
1	`::`	scope	Left-to-right
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	postfix	Left-to-right
3	`++ -- ~ ! sizeof new delete`	unary (prefix)	Right-to-left
	`* &`	indirection and reference (pointers)	
	`+ -`	unary sign operator	
4	`(type)`	type casting	Right-to-left
5	`.* ->*`	pointer-to-member	Left-to-right
6	`* / %`	multiplicative	Left-to-right
7	`+ -`	additive	Left-to-right
8	`<< >>`	shift	Left-to-right
9	`< > <= >=`	relational	Left-to-right
10	`== !=`	equality	Left-to-right
11	`&`	bitwise AND	Left-to-right
12	`^`	bitwise XOR	Left-to-right
13	`|`	bitwise OR	Left-to-right
14	`&&`	logical AND	Left-to-right
15	`||`	logical OR	Left-to-right
16	`?:`	conditional	Right-to-left
17	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment	Right-to-left
18	`,`	comma	Left-to-right



Assignment Operator Deep & Shallow Coping

•**Shallow copy:** when we doesn't define our copy constructor and assignment operator, then compiler defines copy constructor and assignment operator for us and it provides a copying method known as a **shallow copy** ,also known as a **memberwise copy**.

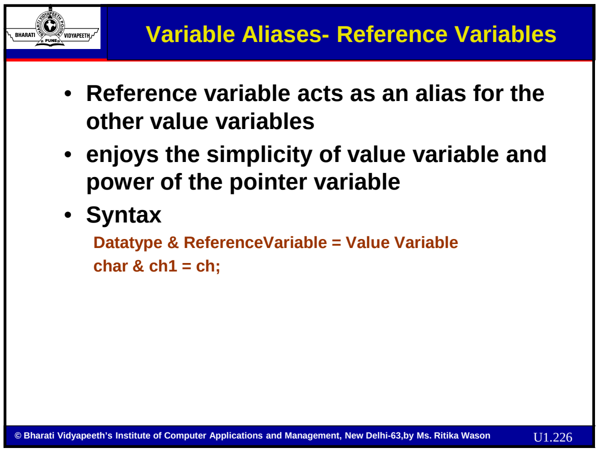
A **shallow copy** copies all of the member variable values. This works efficient if all the member variables are values, but may not work well if member variable point to dynamically allocated memory. The pointer will be copied but the memory it points to will not be copied, the variable in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.



Deep copy:A deep copy copies all member variables, and makes copies of dynamically allocated memory pointed to by the variables. To make a **deep copy**, we have to define our copy constructor and overload the assignment operator.

Requirements for deep copy in the class

- A destructor is required to delete the dynamically allocated memory.
- A copy constructor is required to make a copy of the dynamically allocated memory.
- An overloaded assignment operator is required to make a copy of the dynamically allocated memory.

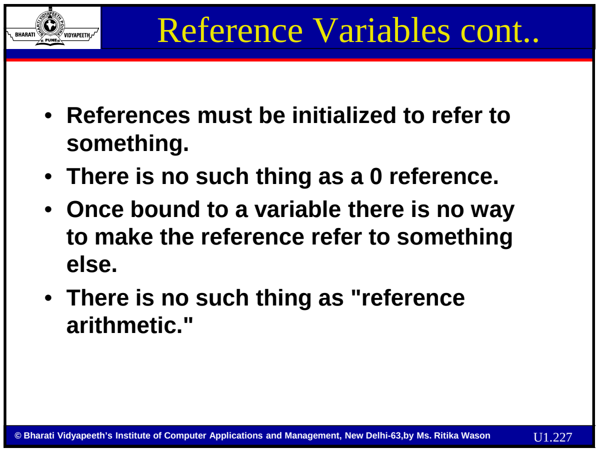


Variable Aliases- Reference Variables

- Reference variable acts as an alias for the other value variables
- enjoys the simplicity of value variable and power of the pointer variable
- Syntax

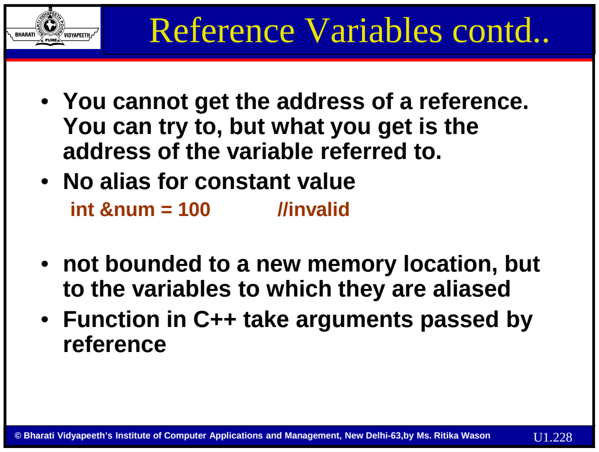
  Datatype & ReferenceVariable = Value Variable

  char & ch1 = ch;



Reference Variables cont..

- References must be initialized to refer to something.
- There is no such thing as a 0 reference.
- Once bound to a variable there is no way to make the reference refer to something else.
- There is no such thing as "reference arithmetic."

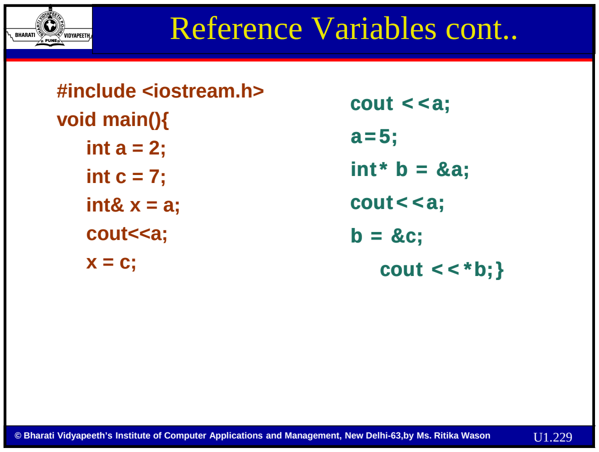


Reference Variables contd..

- You cannot get the address of a reference. You can try to, but what you get is the address of the variable referred to.
- No alias for constant value

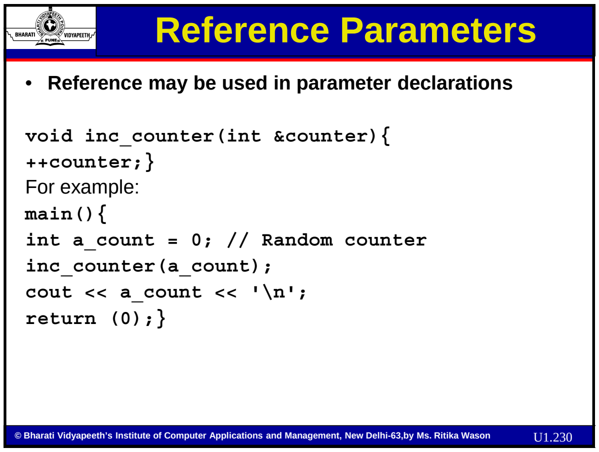
  int &num = 100   //invalid

- not bounded to a new memory location, but to the variables to which they are aliased
- Function in C++ take arguments passed by reference



Reference Variables cont..

```
#include <iostream.h>
void main(){
     int a = 2;
     int c = 7;
     int& x = a;
     cout<<a;
     x = c;
cout <<a;
a=5;
int* b = &a;
cout<<a;
b = &c;
     cout <<*b;}
```



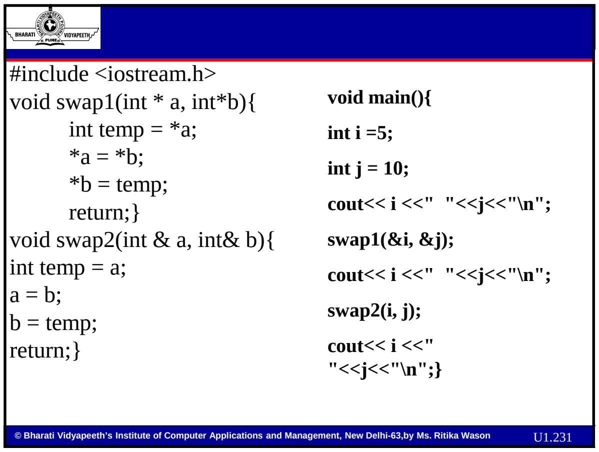
Reference Parameters

- Reference may be used in parameter declarations

```
void inc_counter(int &counter){
++counter;}
```

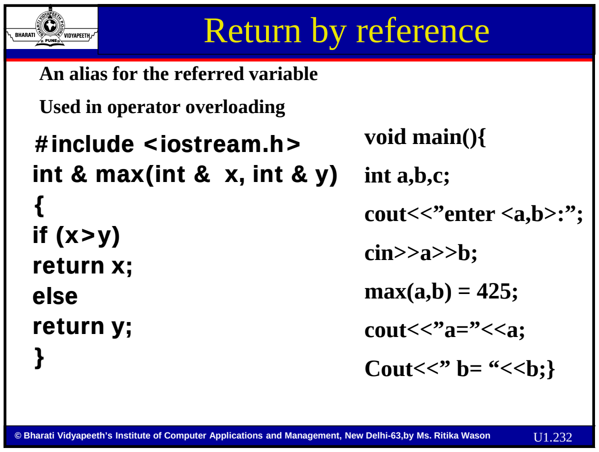
For example:

```
main(){
int a_count = 0; // Random counter
inc_counter(a_count);
cout << a_count << '\n';
return (0);}
```



```
#include <iostream.h>
void swap1(int * a, int*b){
          int temp = *a;
          *a = *b;
          *b = temp;
          return;}
void swap2(int & a, int& b){
int temp = a;
a = b;
b = temp;
return;}
```

```
void main(){
int i =5;
int j = 10;
cout<< i <<"  "<<j<<"\n";
swap1(&i, &j);
cout<< i <<"  "<<j<<"\n";
swap2(i, j);
cout<< i <<"
"<<j<<"\n";}
```



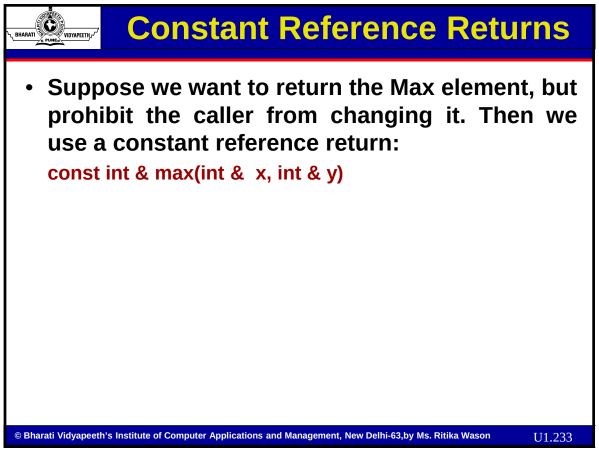
Return by reference

An alias for the referred variable

Used in operator overloading

```
#include <iostream.h>
int & max(int &  x, int & y)
{
if (x>y)
return x;
else
return y;
}
```

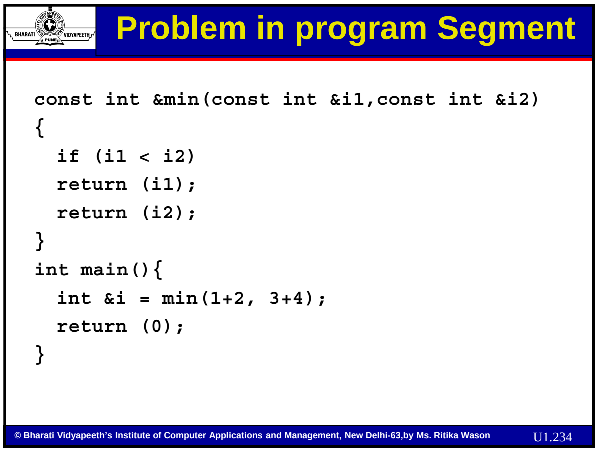
```
void main(){
int a,b,c;
cout<<”enter <a,b>:”;
cin>>a>>b;
max(a,b) = 425;
cout<<”a=”<<a;
Cout<<” b= “<<b;}
```



Constant Reference Returns

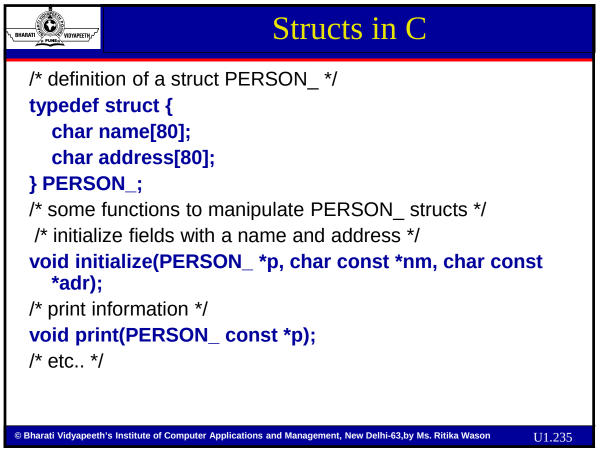
- Suppose we want to return the Max element, but prohibit the caller from changing it. Then we use a constant reference return:

  const int & max(int & x, int & y)



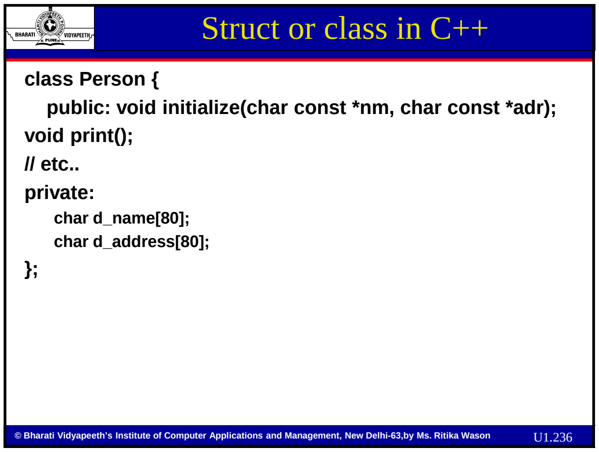
Problem in program Segment

```
const int &min(const int &i1,const int &i2)
{
  if (i1 < i2)
  return (i1);
  return (i2);
}
int main(){
  int &i = min(1+2, 3+4);
  return (0);
}
```



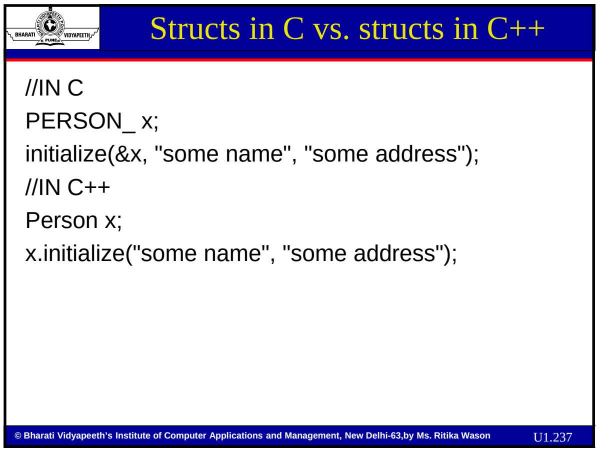
Structs in C

```
/* definition of a struct PERSON_ */
typedef struct {
    char name[80];
    char address[80];
} PERSON_;
/* some functions to manipulate PERSON_ structs */
 /* initialize fields with a name and address */
void initialize(PERSON_ *p, char const *nm, char const
    *adr);
/* print information */
void print(PERSON_ const *p);
/* etc.. */
```



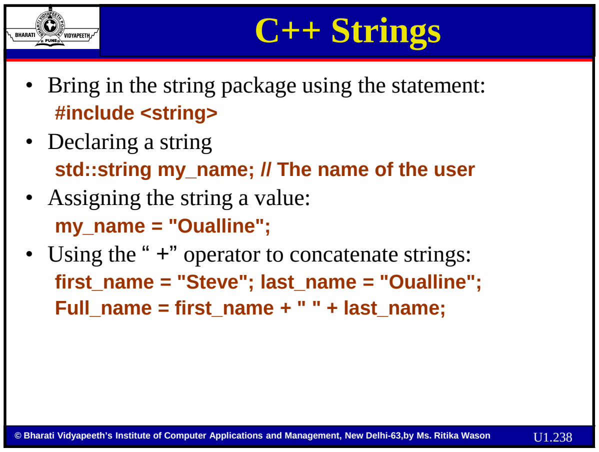
Struct or class in C++

```
class Person {
    public: void initialize(char const *nm, char const *adr);
void print();
// etc..
private:
      char d_name[80];
      char d_address[80];
};
```



Structs in C vs. structs in C++

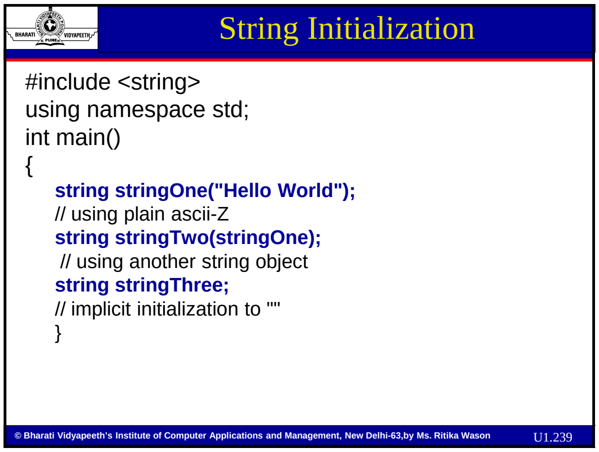
```
//IN C
PERSON_ x;
initialize(&x, "some name", "some address");
//IN C++
Person x;
x.initialize("some name", "some address");
```



C++ Strings

- Bring in the string package using the statement:
  #include <string>
- Declaring a string
  std::string my_name; // The name of the user
- Assigning the string a value:
  my_name = "Oualline";
- Using the “ +” operator to concatenate strings:
  first_name = "Steve"; last_name = "Oualline";
  Full_name = first_name + " " + last_name;

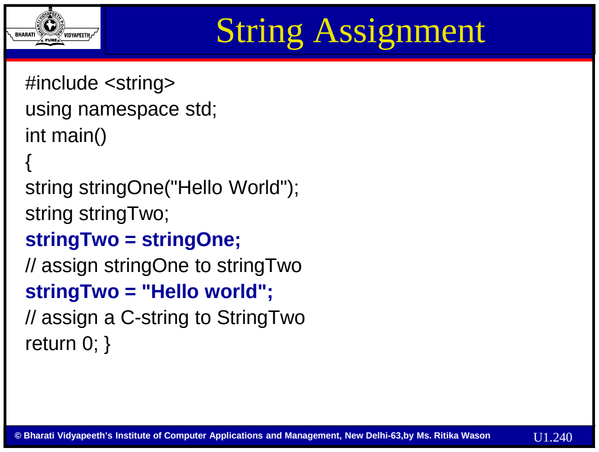
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.238



String Initialization

```
#include <string>
using namespace std;
int main()
{
     string stringOne("Hello World");
     // using plain ascii-Z
     string stringTwo(stringOne);
      // using another string object
     string stringThree;
     // implicit initialization to ""
     }
```

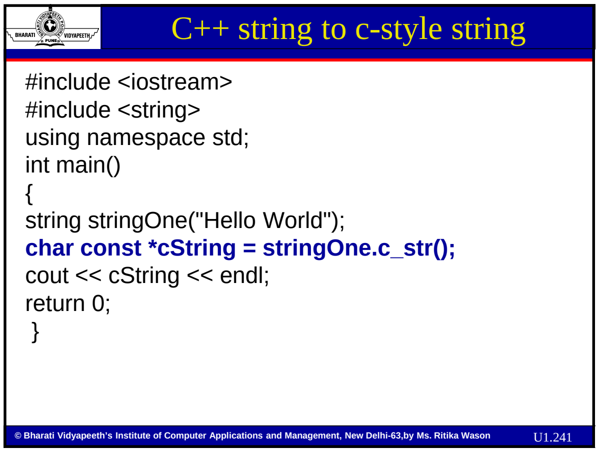
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.239



String Assignment

```
#include <string>
using namespace std;
int main()
{
string stringOne("Hello World");
string stringTwo;
stringTwo = stringOne;
// assign stringOne to stringTwo
stringTwo = "Hello world";
// assign a C-string to StringTwo
return 0; }
```

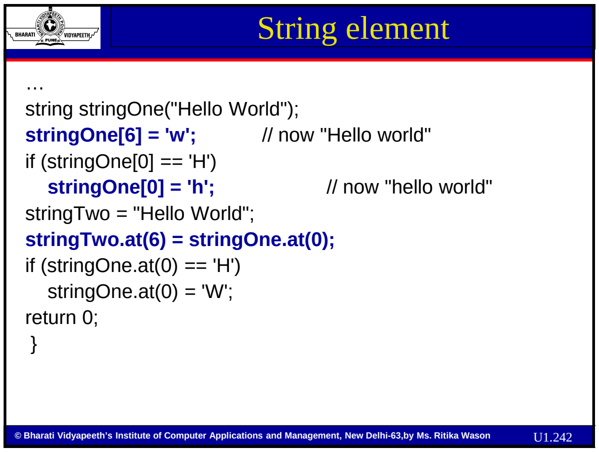
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.240



C++ string to c-style string

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string stringOne("Hello World");
char const *cString = stringOne.c_str();
cout << cString << endl;
return 0;
 }
```

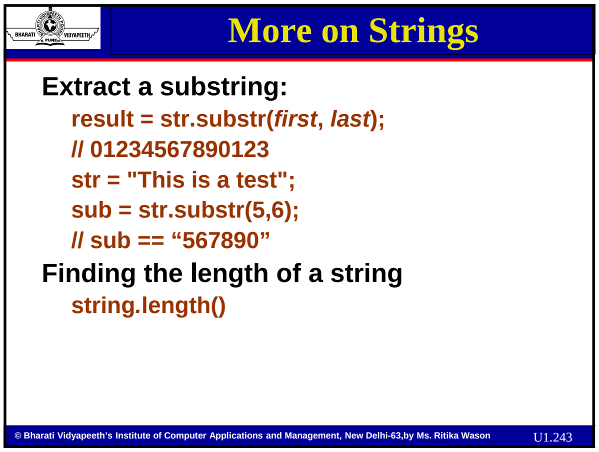
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.241



String element

```
…
string stringOne("Hello World");
stringOne[6] = 'w';            // now "Hello world"
if (stringOne[0] == 'H')
    stringOne[0] = 'h';                // now "hello world"
stringTwo = "Hello World";
stringTwo.at(6) = stringOne.at(0);
if (stringOne.at(0) == 'H')
    stringOne.at(0) = 'W';
return 0;
 }
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.242



More on Strings

Extract a substring:

```
result = str.substr(first, last);
// 01234567890123
str = "This is a test";
sub = str.substr(5,6);
// sub == “567890”
```

Finding the length of a string

```
string.length()
```

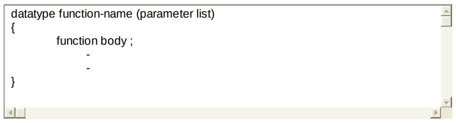
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.243

Functions

- A function is a named unit of a group of program statements. This unit can be invoked from other parts of the program.
- A programmer can solve a simple problem in C++ with a single function but difficult and complex problems can be decomposed into sub-problems, each of which can be either coded directly or further decomposed.
- Decomposing difficult problems, until they are directly code-able as single C++ functions, is a software engineering method of stepwise refinement.
- These functions are combined into other functions and are ultimately used in **main()** to solve the original problem.
- In C++, a function must be defined before it can be used any where in the program.

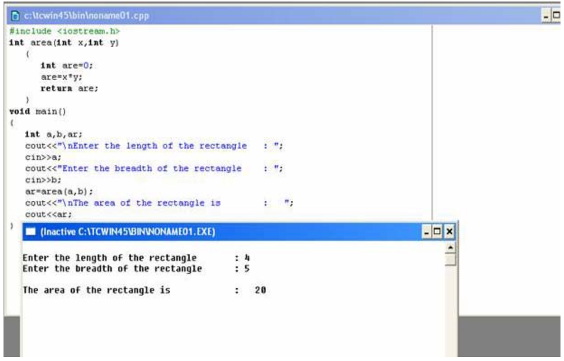
Functions

- The general function definition is as given below:



```
datatype function-name (parameter list)
{
        function body ;
                -
                -
}
```

- where **datatype** specifies the type of value the function returns (e.g.: int, char, float, double, user-defined). If no **datatype** is mentioned, the compiler assumes it as default integer type.
- The parameter list is also known as the arguments or signature of the function, which are the variables that are sent to the function to work on.
- If the parameter list is empty, the compiler assumes that the function does not take any arguments.



c:\tcwin45\bin\noname01.cpp

```
#include <iostream.h>
int area(int x,int y)
   {
      int are=0;
      are=x*y;
      return are;
   }
void main()
{
   int a,b,ar;
   cout<<"\nEnter the length of the rectangle   : ";
   cin>>a;
   cout<<"Enter the breadth of the rectangle    : ";
   cin>>b;
   ar=area(a,b);
   cout<<"\nThe area of the rectangle is        :   ";
   cout<<ar;
}
```

(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
Enter the length of the rectangle       : 4
Enter the breadth of the rectangle      : 5

The area of the rectangle is            :   20
```

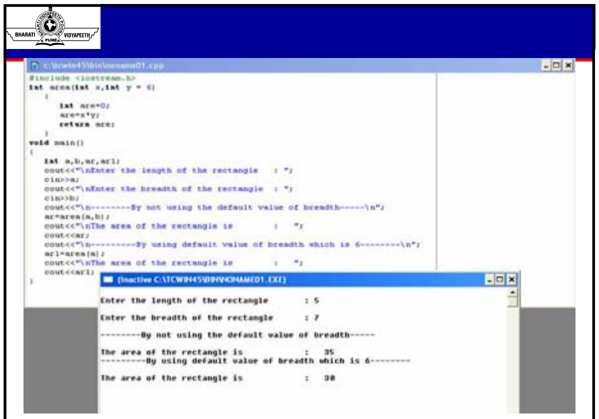
Functions

- The above program allows the user to input two integer values. It then lets the program compute the result by sending both the values to a function, by the name **area()**.
- A function prototype is a declaration of the function that tells the program about the type of value returned by the function and the number & type of arguments passed.
- A function definition is automatically a declaration.

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

Default Arguments

- C++ allows us to assign default values to function parameters, which are used in case a matching argument is not passed in the function call statement.
- The default values must to be specified at the time of definition.



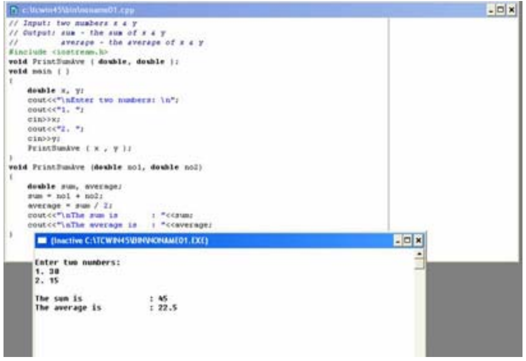
c:\tcwin45\bin\noname01.cpp
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
#include <iostream.h>
int area(int x,int y = 6)
    {
        int are=0;
        are=x*y;
        return are;
    }
void main()
{
    int a,b,ar,ar1;
    cout<<"\nEnter the length of the rectangle   : ";
    cin>>a;
    cout<<"\nEnter the breadth of the rectangle  : ";
    cin>>b;
    cout<<"\n--------By not using the default value of breadth-----\n";
    ar=area(a,b);
    cout<<"\nThe area of the rectangle is          :    ";
    cout<<ar;
    cout<<"\n---------By using default value of breadth which is 6--------\n";
    ar1=area(a);
    cout<<"\nThe area of the rectangle is          :    ";
    cout<<ar1;
}
```

```
Enter the length of the rectangle       : 5

Enter the breadth of the rectangle      : 7

--------By not using the default value of breadth-----

The area of the rectangle is            :   35
---------By using default value of breadth which is 6--------

The area of the rectangle is            :   30
```

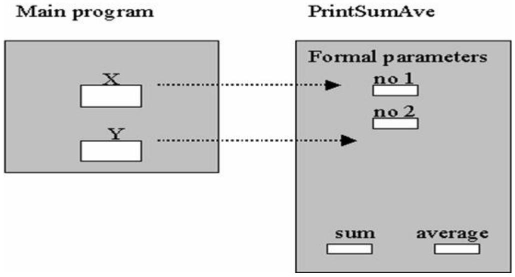
Pass-by-Value

- A function can be invoked in two manners viz. pass-by-value and pass-by-reference . The pass-by-value method copies the actual parameters into the formal parameters, that is, the function makes its own copy of the argument and then uses them.
- The main benefit of call-by-value method is that the original copy of the parameters remains intact and no alteration by the function is reflected on the original variable. All execution is done on the formal parameters; hence, it insures the safety of the data.
- The drawback of call-by-value is that a separate copy of the arguments is used by the function, which occupies some memory space thus increasing the size of the program.


c:\tcwin45\bin\noname01.cpp
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
// Input: two numbers x & y
// Output: sum - the sum of x & y
//          average - the average of x & y
#include <iostream.h>
void PrintSumAve ( double, double );
void main ( )
{
    double x, y;
    cout<<"\nEnter two numbers: \n";
    cout<<"1. ";
    cin>>x;
    cout<<"2. ";
    cin>>y;
    PrintSumAve ( x , y );
}
void PrintSumAve (double no1, double no2)
{
    double sum, average;
    sum = no1 + no2;
    average = sum / 2;
    cout<<"\nThe sum is        : "<<sum;
    cout<<"\nThe average is    : "<<average;
}
```

```
Enter two numbers:
1. 30
2. 15

The sum is              : 45
The average is          : 22.5
```

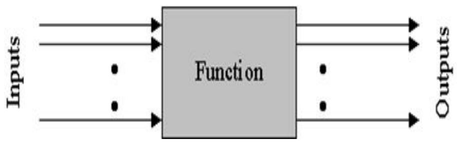


The figure shows that the original parameters are copied into the formal parameters in the PrintSumAve() function and then executed in the function body.

Call-By-Reference

- The call by reference uses a different mechanism. In place of passing value to the function, which is called, a reference to the original variable is passed.
- A reference is an alias for the predefined variable. That is, the value of that variable can be accessed by using any of the two: the original or the reference variable name.
- When a function is called by reference, then the formal parameters become reference to the actual parameters in the calling function. This means that, in call by reference method, the called function does not create its own copy of the original values, rather, it refers to the original values only by different names.
- This called function works with the original data and any change in the value is reflected back to the data.

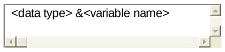
Call-By-Reference



To write a function, which returns multiple output, we have to pass parameters by reference.

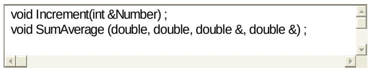
Call-By-Reference in Functions

We use **&** to denote a parameter that is passed by reference:



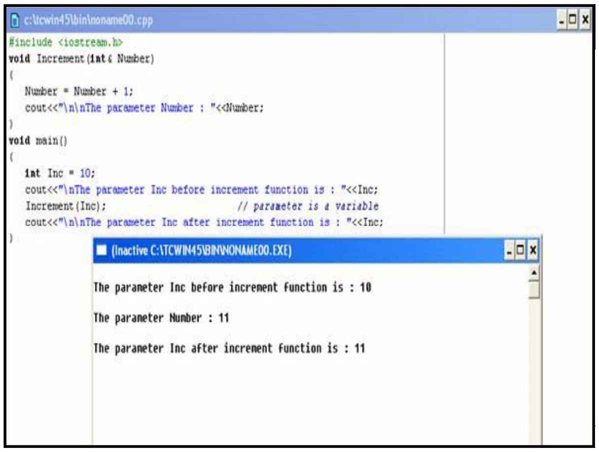
```
<data type> &<variable name>
```

Example:



```
void Increment(int &Number) ;
void SumAverage (double, double, double &, double &) ;
```

- The address (reference) of the actual variable is passed to the function, instead of its value.
- If the function changes the parameter's value, the change is reflected back in the actual variables in the called function, since they share the same memory location.

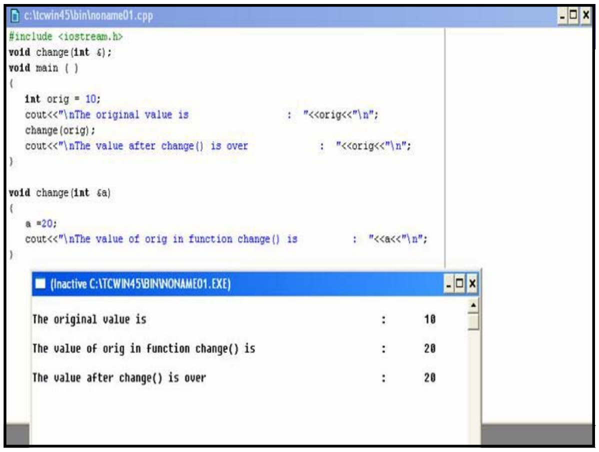


c:\tcwin45\bin\noname00.cpp

```
#include <iostream.h>
void Increment(int& Number)
{
   Number = Number + 1;
   cout<<"\n\nThe parameter Number : "<<Number;
}
void main()
{
   int Inc = 10;
   cout<<"\nThe parameter Inc before increment function is : "<<Inc;
   Increment(Inc);                    // parameter is a variable
   cout<<"\n\nThe parameter Inc after increment function is : "<<Inc;
}
```

(Inactive C:\TCWIN45\BIN\NONAME00.EXE)

```
The parameter Inc before increment function is : 10

The parameter Number : 11

The parameter Inc after increment function is : 11
```



c:\tcwin45\bin\noname01.cpp

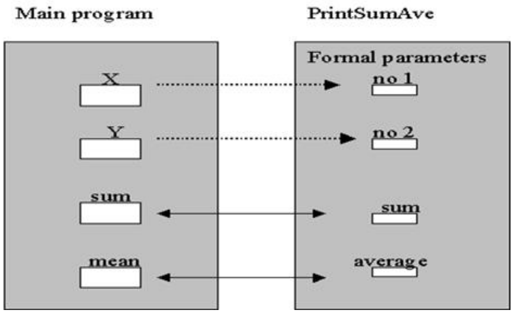
```
#include <iostream.h>
void change(int &);
void main ( )
{
   int orig = 10;
   cout<<"\nThe original value is              :  "<<orig<<"\n";
   change(orig);
   cout<<"\nThe value after change() is over         :  "<<orig<<"\n";
}

void change(int &a)
{
   a =20;
   cout<<"\nThe value of orig in function change() is        :  "<<a<<"\n";
}
```

(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

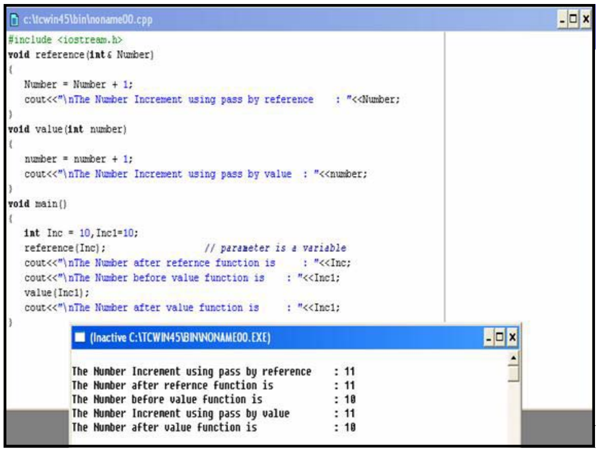
```
The original value is                              :      10

The value of orig in function change() is          :      20

The value after change() is over                   :      20
```

Call by Reference

- In the above program, the function **change()** refers to the original value of **orig,** which is 10, by its reference **a**.
- The same memory location is referred to by **orig** in **main()** and by **a** in **change().**
- Hence the change in **a** , by assigning value20, is reflected in **orig** also.



Pass-by-value Vs Pass-by-reference



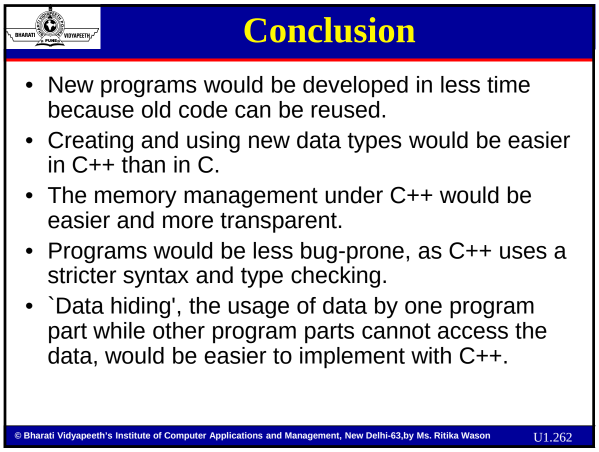
```
#include <iostream.h>
void reference(int& Number)
{
   Number = Number + 1;
   cout<<"\nThe Number Increment using pass by reference    : "<<Number;
}
void value(int number)
{
   number = number + 1;
   cout<<"\nThe Number Increment using pass by value  : "<<number;
}
void main()
{
   int Inc = 10,Inc1=10;
   reference(Inc);                    // parameter is a variable
   cout<<"\nThe Number after refernce function is     : "<<Inc;
   cout<<"\nThe Number before value function is    : "<<Inc1;
   value(Inc1);
   cout<<"\nThe Number after value function is     : "<<Inc1;
}
```

```
The Number Increment using pass by reference    : 11
The Number after refernce function is           : 11
The Number before value function is             : 10
The Number Increment using pass by value        : 11
The Number after value function is              : 10
```

Pass by Value / Pass by Reference

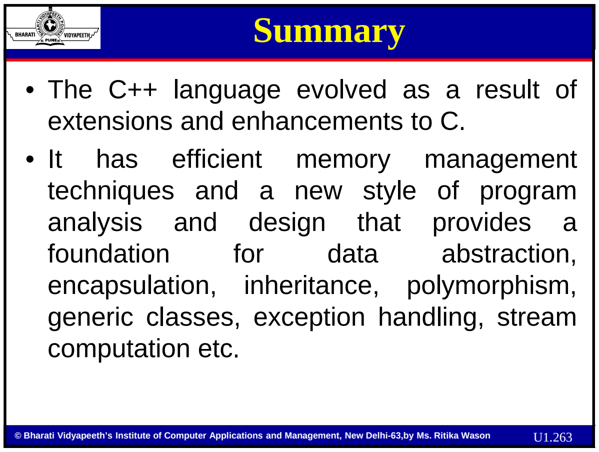
This program uses two functions, one using pass-by-value method and the other using pass-by-reference method.

This shows how the value of **number** does not change in the main function, when it is evaluated using pass-by-value method and on the other hand it changes when implemented using pass-by-reference method.



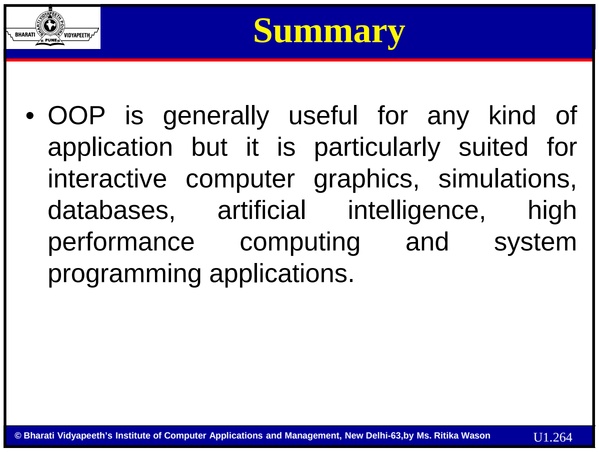
Conclusion

- New programs would be developed in less time because old code can be reused.
- Creating and using new data types would be easier in C++ than in C.
- The memory management under C++ would be easier and more transparent.
- Programs would be less bug-prone, as C++ uses a stricter syntax and type checking.
- `Data hiding', the usage of data by one program part while other program parts cannot access the data, would be easier to implement with C++.



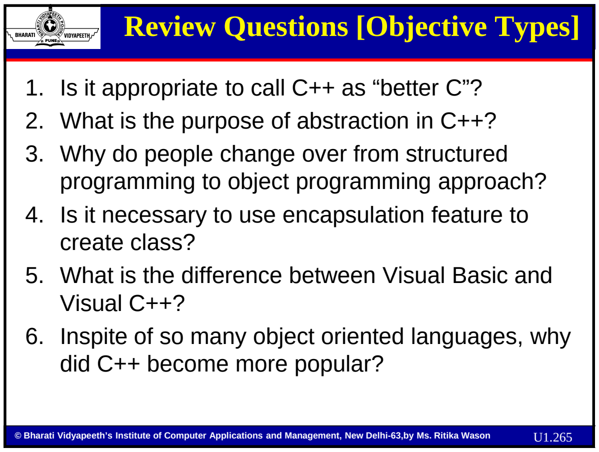
Summary

- The C++ language evolved as a result of extensions and enhancements to C.
- It has efficient memory management techniques and a new style of program analysis and design that provides a foundation for data abstraction, encapsulation, inheritance, polymorphism, generic classes, exception handling, stream computation etc.



Summary

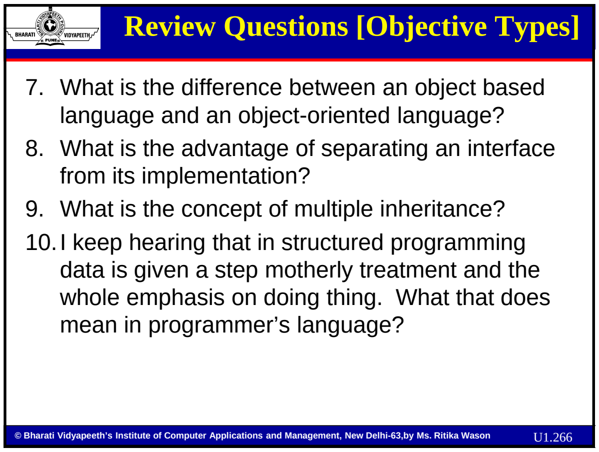
- OOP is generally useful for any kind of application but it is particularly suited for interactive computer graphics, simulations, databases, artificial intelligence, high performance computing and system programming applications.



Review Questions [Objective Types]

1. Is it appropriate to call C++ as “better C”?
2. What is the purpose of abstraction in C++?
3. Why do people change over from structured programming to object programming approach?
4. Is it necessary to use encapsulation feature to create class?
5. What is the difference between Visual Basic and Visual C++?
6. Inspite of so many object oriented languages, why did C++ become more popular?

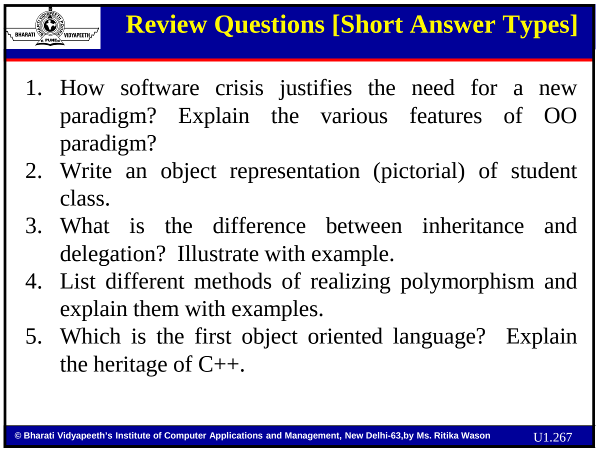
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.265



Review Questions [Objective Types]

7. What is the difference between an object based language and an object-oriented language?
8. What is the advantage of separating an interface from its implementation?
9. What is the concept of multiple inheritance?
10. I keep hearing that in structured programming data is given a step motherly treatment and the whole emphasis on doing thing. What that does mean in programmer’s language?

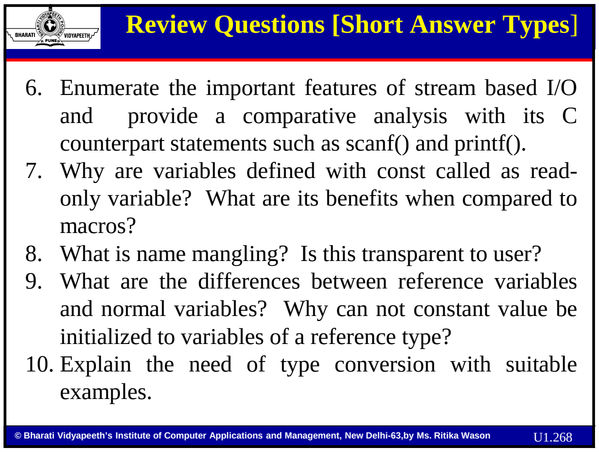
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.266



Review Questions [Short Answer Types]

1. How software crisis justifies the need for a new paradigm? Explain the various features of OO paradigm?
2. Write an object representation (pictorial) of student class.
3. What is the difference between inheritance and delegation? Illustrate with example.
4. List different methods of realizing polymorphism and explain them with examples.
5. Which is the first object oriented language? Explain the heritage of C++.

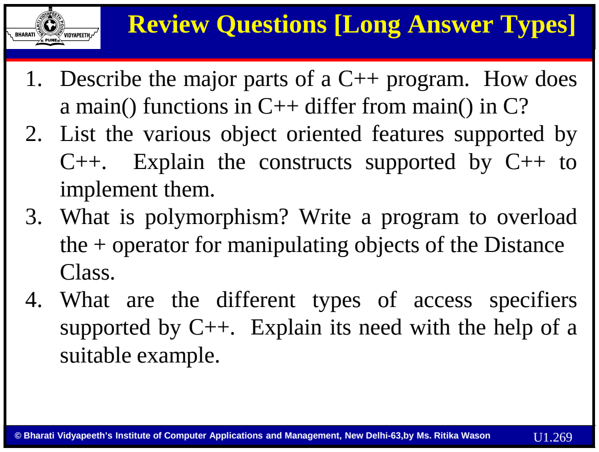
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.267



Review Questions [Short Answer Types]

6. Enumerate the important features of stream based I/O and provide a comparative analysis with its C counterpart statements such as scanf() and printf().
7. Why are variables defined with const called as read-only variable? What are its benefits when compared to macros?
8. What is name mangling? Is this transparent to user?
9. What are the differences between reference variables and normal variables? Why can not constant value be initialized to variables of a reference type?
10. Explain the need of type conversion with suitable examples.

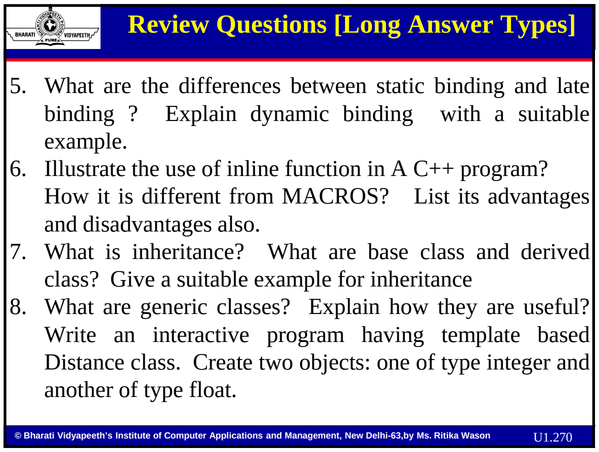
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.268



Review Questions [Long Answer Types]

1. Describe the major parts of a C++ program. How does a main() functions in C++ differ from main() in C?
2. List the various object oriented features supported by C++. Explain the constructs supported by C++ to implement them.
3. What is polymorphism? Write a program to overload the + operator for manipulating objects of the Distance Class.
4. What are the different types of access specifiers supported by C++. Explain its need with the help of a suitable example.

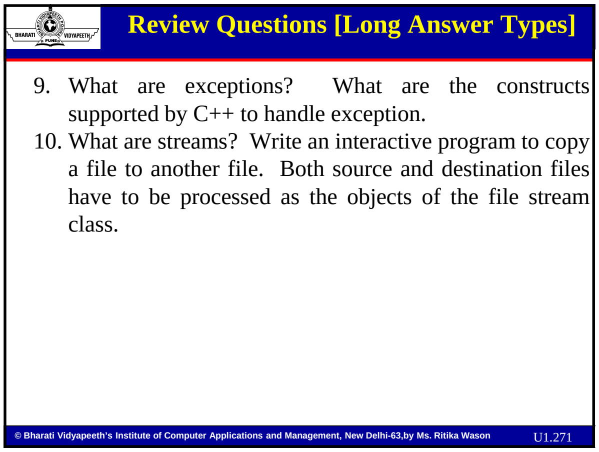
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.269



Review Questions [Long Answer Types]

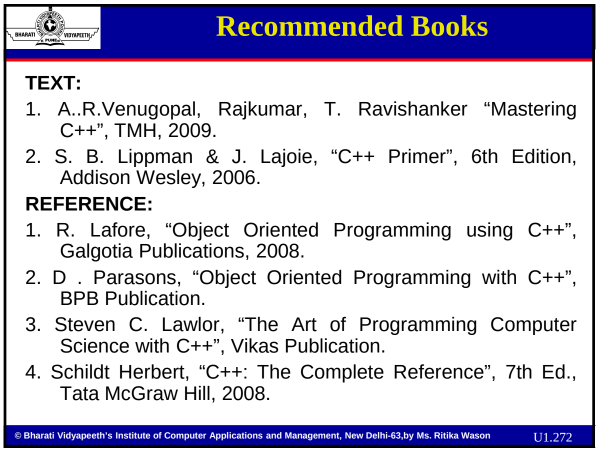
5. What are the differences between static binding and late binding ? Explain dynamic binding with a suitable example.
6. Illustrate the use of inline function in A C++ program? How it is different from MACROS? List its advantages and disadvantages also.
7. What is inheritance? What are base class and derived class? Give a suitable example for inheritance
8. What are generic classes? Explain how they are useful? Write an interactive program having template based Distance class. Create two objects: one of type integer and another of type float.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,by Ms. Ritika Wason U1.270



Review Questions [Long Answer Types]

9. What are exceptions? What are the constructs supported by C++ to handle exception.
10. What are streams? Write an interactive program to copy a file to another file. Both source and destination files have to be processed as the objects of the file stream class.



Recommended Books

TEXT:

1. A..R.Venugopal, Rajkumar, T. Ravishanker "Mastering C++", TMH, 2009.
2. S. B. Lippman & J. Lajoie, "C++ Primer", 6th Edition, Addison Wesley, 2006.

REFERENCE:

1. R. Lafore, "Object Oriented Programming using C++", Galgotia Publications, 2008.
2. D . Parasons, "Object Oriented Programming with C++", BPB Publication.
3. Steven C. Lawlor, "The Art of Programming Computer Science with C++", Vikas Publication.
4. Schildt Herbert, "C++: The Complete Reference", 7th Ed., Tata McGraw Hill, 2008.